Módulo 9

Banco de Dados

Lição 7

Gerenciamento de Transações

Versão 1.0 - Fev/2009

Autor
Ma. Rowena C. Solamo

Equipe
Rommel Feria
Rick Hillegas
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/60/relnotes.htm

Java™ DB System Requirements

Java™ DB is supported on the Solaris, Linux and Windows operating systems and Sun Java 1.4 or later.

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alberto Ivo da Costa Vieira
Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
Angelo de Oliveira
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Carlos Hilner Ferreira Costa
Daniel Noto Paiva
Daniel Wildt
Denis Mitsuo Nakasaki
Fábio Antonio Ferreira
Givailson de Souza Neves
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Vianney Barrozo Costa
José Francisco Baronio da Costa
Kleberth Bezerra Galvão dos Santos
Luiz Fernandes de Oliveira Junior
Maria Carolina Ferreira da Silva
Maricy Caregnato
Mauricio da Silva Marinho
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw
Seire Pareja
Sergio Terzella
Thiago Magela Rodrigues Dias

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Um sistema de banco de dados relacional deve prover três importantes serviços para garantir que a base de dados seja confiável, e se mantenha em um estado consistente. Esse estado deverá ser mantido mesmo na presença de falhas de hardware, software e quando múltiplos usuários estejam acessando a base de dados. Este capítulo lida com esses três serviços principais. São eles: suporte a transação, recuperação de base de dados e controle de concorrência.

- Suporte a Transação
  - Propriedades ACID
  - Arquitetura de Base de Dados
- Controle de Concorrência
  - Serialização e Recuperabilidade
  - Técnicas de Controle de Concorrência
    - Paralisações (*Deadlocks*)
    - Etiquetas de tempo (*Timestamping*)
    - Técnicas Otimistas
  - Granularidade de Itens de Dados
- Recuperação de Base de Dados
  - Transações e Recuperação
  - Facilidades de Recuperação
  - Técnicas de Recuperação
- Gerenciamento de Transação JavaDB

Ao final desta lição, o estudante será capaz de:

- Conhecer como o banco de dados faz a gestão das transações.
- Descrever os três importantes serviços necessários para saber que um banco de dados é confiável e permanece num estado consistente.
- Os conceito de transações, de controle de concomitância e de recuperação de banco de dados.
- Realizar técnicas de recuperação.

2. Suporte a Transação

Uma ***transação*** é uma série de operações sobre uma base de dados que o usuário necessita que se completem totalmente ou sejam canceladas. É uma ação ou série de ações que são executadas por um único usuário ou programa aplicativo, que acesse ou modifique o conteúdo da base de dados. É uma unidade lógica de trabalho que pode ser um programa inteiro ou parte de um programa que pode conter comandos SQL.

No contexto da base de dados, a execução de um programa aplicativo pode ser vista como uma série de transações ocorrendo com o processamento não orientado a base de dados. Para melhor entender o conceito de transação, considere duas relações na base de dados `Loja Orgânica` conforme especificadas a seguir:

```
INVENTORY(store_no, item_code, qtd, level);
ORDER_DETAILS (order_no, line_no, qtd_ordered, item, unit_price);
```

Uma transação simples sobre esta base de dados é atualizar a quantidade de um item em particular de uma determinada loja para repor seu estoque. Para representar a leitura na base de dados do item de dado x, utilizaremos a notação:

```
read(x)
```

Para representar a escrita na base de dados do item de dado x, utilizaremos a notação:

```
write(x)
```

Para representar a atualização da quantidade de um item para repor o estoque, uma transação pode ser escrita da seguinte maneira:

```
read(item_code=x and store_no=y, qtd)
qtd = qtd + additional_stock
write(item_code=x and store_no=y, qtd)
```

Uma transação mais complexa para realizar um pedido inserindo um registro de *Order_Detail* é mostrada a seguir:

```
insert_Order_Details(order_no=a, line_no=b, qtd_ordered=c, item_code=x)
begin
    read_inventory(item_code=x and store_no=y, qtd)
    qtd = qtd – quanity_ordered
    write_inventory(item_code=x and store_no=y, qtd)
    write_Order_Details(order_no=a, line_no=b, qtd_ordered=c, item_code=x)
end
```

Neste exemplo, antes de registrar a venda, a informação de inventário do item é atualizada pela subtração da quantidade demandada em estoque. A seguir, o registro que correspondente à venda será inserido.

Uma transação pode ter um dos seguintes desfechos. Pode ser realizada (***commit***), ou seja, a transação foi completada com sucesso e a base de dados chegou a um novo estado consistente. Ou pode ser desfeita (***rollback*** ou ***undone***), ou seja, a transação não foi executada com sucesso ou abortou, a base de dados foi restituída ao estado consistente que tinha antes do início da transação.

Uma transação realizada não pode ser desfeita. Se uma transação foi considerada errônea, outra transação deve ser executada para reverter o seu efeito. Este tipo de transação é conhecida como transação de compensação. Contudo, uma transação abortada que foi desfeita pode ser reiniciada e, dependendo da causa da falha, pode executar com sucesso e ser realizada mais tarde.

O sistema de banco de dados relacional não tem uma forma inerente de saber quais comandos estão agrupados para formar uma transação. Deve prover um meio para permitir aos usuários indicarem as fronteiras da transação. Em alguns sistemas as palavras chave BEGIN TRANSACTION, COMMIT e ROLLBACK (ou seus equivalentes) são utilizadas. Para JavaDB, a JDBC

API é usada para definir as transações.

2.1. Propriedades de Transações

Existem as seguintes propriedades que são obrigatórios para as transações. A sigla ACID é utilizada para representar estas propriedades.

1. ***Atomicidade***. É conhecida como a propriedade do 'tudo ou nada'. Uma transação é uma unidade singular que ou é executada inteiramente, ou não é executada de forma nenhuma.
2. ***Consistência***. Uma transação deve transformar a base de dados de um estado consistente para outro estado consistente.
3. ***Isolamento***. As transações são executadas independentes umas das outras. Em outras palavras, os efeitos parciais de transações incompletas não deverão ser visíveis para outras transações.
4. ***Durabilidade***. Os efeitos de uma transação completada com sucesso (committed) são gravadas de forma permanente na base de dados, e não podem ser perdidas por causa de uma falha subsequente.

2.2. Arquitetura de Base de Dados

A Figura 1 mostra um diagrama da arquitetura de base de dados com quatro módulos que lidam com transações, controle de concorrência e recuperação. O gerente da transação é o responsável pela coordenação das transações sobre os programas aplicativos. Se comunica com o plano de execuçãodor que é o módulo responsável pela implementação de uma estratégia de controle de concorrência em particular. O plano de execuçãodor também é conhecido como o Gerente de Bloqueio, caso o controle de concorrência seja baseado em bloqueios (*lock-based*). O propósito do plano de execuçãodor é maximizar a concorrência sem permitir que transações que estão executando concorrentemente interfiram umas com as outras. Isto é feito de modo que a integridade e a consistência da base de dados não sejam comprometidas.

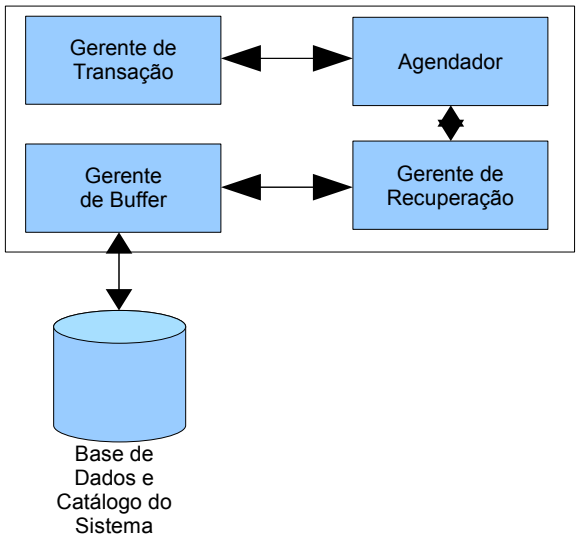

Figura 1: Subsistema de Transações de uma Arquitetura de Base de Dados

Se uma falha ocorrer durante a transação, a base de dados estará em um estado inconsistente. É tarefa do Gerente de Recuperação garantir que a base de dados seja restaurada ao estado que estava antes do início da transação; retornando a base de dados para um estado consistente. Por último, o gerente de buffer é responsável pela transferência de dados entre o dispositivo de armazenamento de disco e a memória principal.

3. Controle de Concorrência

Controle de Concorrência é o processo de gerenciar operações simultâneas sobre a base de dados sem que haja interferencia entre elas. Discutiremos nesta seção os problemas que surgem do acesso concorrente e as técnicas que podem ser empregadas para evitá-los.

3.1. Por que precisamos de controle de concorrência?

Um SGBD deve habilitar vários usuários a acessar dados compartilhados concorrentemente. O acesso concorrente é relativamente fácil de ser gerenciado se todos os usuários estão executando operações de leitura em dados compartilhados, pois não há como uns interferirem com os outros. Contudo, quando dois ou mais usuários estão acessando a base de dados simultaneamente, e pelo menos um deles está modificando os dados, interferências que podem resultar em inconsistências podem ocorrer.

Contudo, duas transações podem estar perfeitamente corretas por si só, mas o entrelaçamento de operações pode produzir resultados incorretos, comprometendo a integridade e a consistência da base de dados. Nesta seção, três exemplos de problemas potenciais causados pela concorrência são discutidos abaixo. São eles, o *Problema da Atualização Perdida*, o *Problema da Dependência Incompleta*, e o *Problema da Análise de Consistência*.

Problema da Atualização Perdida

Onde uma operação de atualização que aparentemente foi realizada com sucesso por um usuário pode ser sobrescrita por outra operação de atualização de um outro usuário. Considere a transação a seguir onde assumimos que existem duas transações T_1 e T_2 que tentam atualizar a quantidade disponível para um determinado estoque. T_1 adiciona à quantidade disponível para repor o estoque enquanto T_2 subtrai da quantidade disponível quando um pedido é feito.

Tempo	**T_1**	**T_2**	**$quantidade_x$**
t1		begin	150
t2	begin	read(qtd_x)	150
t3	read(qtd_x)	qtd_x = qtd_x +100	150
t4	qtd_x = qtd_x - 10	write(qtd_x)	250
t5	write(qtd_x)	commit	140
t6	commit		140

Transacao 1: Problema da Atualização Perdida

Assumimos então que T_1 e T_2 se iniciem simultaneamente. Ambas lêem a coluna da quantidade (150). T_1 adiciona o valor 100 e armazena 250 na base de dados. Enquanto T_2 subtrai o valor 10, e armazena este novo valor (140) na base de dados. Esta atualização sobrescreve a atualização anterior e assim é perdida a transação feita por T_1. A perda da atualização de T_1 é evitada impedindo T_2 de ler o valor da quantidade até que T_1 tenha completado.

Problema da Dependência Incompleta

Ocorre quando é permitido à transação visualizar resultados intermediários de outra transação antes que esta tenha se completado. A tabela a seguir ilustra este problema:

Tempo	**T_1**	**T_2**	**qtd_x**
t1		begin	150
t2		read(qtd_x)	150
t3		qtd_x = qtd_x +100	150
t4	begin	write(qtd_x)	250
t5	read(qtd_x)	...	250
t6	qtd_x = qtd_x - 10	**rollback**	250
t7	write(qtd_x)		240
t8	commit		240

Transação 2: Problema da Dependência Incompleta

Um erro ocorre ao usar o mesmo valor inicial da quantidade (150) como no exemplo anterior.

Quando T_2 atualiza a quantidade para 250, mas aborta a transação, de modo que a quantidade será restaurada para o valor original de 150. Neste ponto T_1 já leu o novo valor da quantidade (250), e o está utilizando como base para a redução, que resulta em um dado incorreto da quantidade disponível (240).

Problema da Análise Inconsistente

Os dois problemas mostrados anteriormente concentram-se em transações que estão atualizando a base de dados e sua interferência pode corromper a base de dados. Contudo, transações que lêem a base de dados também podem produzir resultados imprecisos se elas puderem ler resultados parciais de transações incompletas que são simultaneamente atualizadas e desfeitas. Isto às vezes é conhecido como ***leitura suja*** ou ***leitura irrepetível***.

Este problema ocorre quando uma transação lê diversos valores de uma base de dados, mas uma segunda transação atualiza alguns deles durante a execução da primeira. Como exemplo, suponha que uma transação que faça a sumarização de dados em uma base de dados, como a totalização do número de itens vendidos. Vejamos o seguinte exemplo:

Tempo	T_1	T_2	qtd_x	qtd_y	qtd_z	soma
t1		begin	150	100	50	
t2	begin	sum = 0	150	100	50	0
t3	read(qtd_x)	read(qtd_x)	150	100	50	0
t4	qtd_x = qtd_x-10	sum = sum + qtd_x	150	100	50	0
t5	write(qtd_x)	read(qtd_y)	140	100	50	150
t6	read(qtd_z)	sum = sum + qtd_y	140	100	50	150
t7	qtd_z = qtd_z+10	...	140	100	50	250
t8	write(qtd_z)	...	140	100	60	250
t9	commit	read(qtd_z)	140	100	60	250
t10		sum = sum + qtd_z	140	100	60	310
t11		commit	140	100	60	310

Transação 3: Problema da Análise Inconsistente

Observe que T_1 atualiza a quantidade enquanto T_2 tenta obter o número total de itens vendidos pela loja. Enquanto T_2 está obtendo a soma, T_1 atualiza a quantidade para os itens x e z. Assim, o total da soma é 310 enquanto que a soma real deveria ser 300 (140+100+60).

Este problema pode ser prevenido não permitindo que T_2 leia $\texttt{quantidade}_x$ e $\texttt{quantidade}_z$ até que T_1 tenha completado sua atualização.

3.2. Serialização e Recuperação

O objetivo do protocolo do controle de concorrência é agendar transações de modo a evitar qualquer interferência. Uma solução é permitir que somente uma transação seja executada, ou seja, uma transação deve se completar (*commit*) antes que outra inicie (*begin*). Contudo, o SGBD roda em um ambiente multi-usuário. Então maximizamos o grau de concorrência ou paralelismo no sistema de modo que as transações possam executar sem interferir umas com as outras e possam rodar em paralelo. Transações que acessam e atualizam partes diferentes da base de dados podem ser executadas juntas e sem interferência. O conceito de **serialização** é um meio de melhorar as execuções das transações que sejam garantidas para manter a consistência.

Uma transação é uma sequência de operações que consiste de ações de leitura e escrita na base de dados. É seguida de uma ação de *commit* ou *rollback*. Para prover um controle de concorrência adequado, um **plano de execução** precisa existir para garantir a consistência dos nossos dados. Um plano de execução é uma sequência de operações de transações concorrentes que preserva a ordem das operações para cada transação individual. Mais formalmente:

> Um plano de execução S consiste de uma sequência de operações de um conjunto n de transações T_1, T_2, ..., T_n, sujeitos à restrição de que a ordem das operações para cada transação seja preservada no plano de execução.

Para cada transação T_i no plano de execução S, a ordem das operações em T devem estar no mesmo plano de execução.

Um **plano de execução não-serial** é onde as operações de um conjunto de transações

concorrentes são intercaladas.

Um **plano de execução serial** é onde as operações de cada transação são executadas consecutivamente sem intercalação de operações de outras transações. Se tivermos duas transações T_1 e T_2, a ordem serial seria T_1, e então T_2. Ou poderia ser T_2, e então T_1. Não há interferência entre as transações pois somente uma está executando a qualquer dado momento. Contudo, não há garantia que os resultados de todas as execuções seriais de um dado conjunto de transações sejam idênticos. Como exemplo, na área bancária, a aplicação da taxa de juros antes ou depois de um grande depósito faz diferença.

O objetivo da serialização é encontrar planos de execução não-seriais que permitam que as transações executem sem a interferência entre si; consequentemente, produzindo um estado da base de dados que será reproduzido por uma execução serial. Se um conjunto de transações executa concorrentemente, a plano de execução (não-serial) está correto se produz os mesmos resultados de alguma execução serial. Um plano de execução é conhecido como serializável. É essencial garantir a serialização para prevenir inconsistências de transações concorrentes. Na serialização, a ordem das operações de leitura e escrita é importante. Listamos a seguir alguns princípios sobre a serialização de transações.

- Se duas transações lêem somente dados, não interferem. Portanto, a ordem não é importante.
- Se duas transações lêem ou escrevem em itens de dados separados, não interferem. Portanto, a ordem não é importante.
- Se uma transação escreve em um item de dado e outra lê ou escreve no mesmo item de dado, a ordem de execução é importante.

Considere um plano de execução S_1 como mostrado a seguir. As operações de duas transações T_1 e T_2 executando concorrentemente.

T_1	T_2
begin	
read(qtd_x)	
write(qtd_x)	
	begin
	read(qtd_x)
	write(qtd_x)
read(qtd_y)	
write(qtd_y)	
commit	
	read(qtd_y)
	write(qtd_y)
	commit

Transação 4: plano de execução S_1

Como as operações de escrita em qtd_x em T_2 não conflitam com as operações de leitura subsequentes de qtd_y de T_1, podemos alterar a ordem dessas operações para produzir um plano de execução equivalente S_2 que é mostrada a seguir:

T_1	T_2
begin	
read(qtd_x)	
write(qtd_x)	
	begin
	read(qtd_x)
read(qtd_y)	
	write(qtd_x)
write(qtd_y)	
commit	
	read(qtd_y)
	write(qtd_y)
	commit

Transação 5: plano de execução S_2

Ao mudar a ordem das operações não-conflitantes, produz-se um plano de execução equivalente S_3, que é mostrado a seguir, um plano de execução S_3 é serial. Como S_1 e S_2 são equivalentes a S_3, S_1 e S_2 também são seriais.

T_1	T_2
begin	
read(qtd_x)	
write(qtd_x)	
read(qtd_y)	
write(qtd_y)	
commit	
	begin
	read(qtd_x)
	write(qtd_x)
	read(qtd_y)
	write(qtd_y)
	commit

Transação 6: plano de execução S_3

Este tipo de serialização é conhecido como **Serialização de Conflito** em que um plano de execução serializável ordena quaisquer operações conflitantes do mesmo modo como as execuções seriais. Para testar uma serialização de conflito, podemos gerar um grafo de precedência baseado na restrição de escrita, ou seja, uma transação atualiza um item de dado baseado no seu valor antigo e é primeiramente lido pela transação. O grafo de precedência consiste do seguinte:

- um nó para representar cada transação
- uma aresta direcionada, T_i -> T_j, se T_j lê o valor de um item escrito por T_i
- uma aresta direcionada, T_i -> T_j, se T_j escreve o valor em um item depois que este foi lido por T_i
- Se o grafo de precedência contiver um ciclo, o plano de execução não é serializável de conflito. Considere as transações T_1 e T_2 abaixo. A transação T_1 adiciona mais de 100 unidades a qtd_x e subtrai 100 de qtd_y, enquanto T_2 subtrai 50 de ambos. O grafo de precedência é mostrado na Figura 2.

T_1	T_2
begin	
read(qtd_x)	
$qtd_x = qtd_x + 100$	
	begin
	read(qtd_x)
	$qtd_x = qtd_x - 50$
	write(qtd_x)
	read(qtd_y)
	$qtd_y = qtd_y - 50$
	write(qtd_y)
	commit
read(qtd_y)	
$qtd_y = qtd_y - 100$	
write(qtd_y)	
commit	

Transação 7: Duas Transações Concorrentes

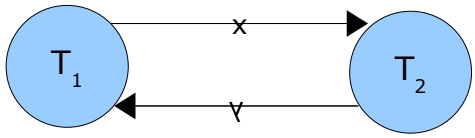

Figura 2: Grafo de Precedência da transação (Duas Transações Concorrentes)

Existe um ciclo e, portanto, não serializável de conflito. Uma serialização menos restritiva do que a serialização de conflito é a **Serialização de Visão**. Dois plano de execuções S_1 e S_2 consistem

das mesmas operações de n transações, T_1, T_2,...,T_n Serão equivalentes de visão se as seguintes três condições forem satisfeitas:

1. Para cada item de dado x, se a transação T_i ler o valor inicial de x no plano de execução S_1, então a transação T_i também deve ler o valor inicial de x no plano de execução S_2.
2. Para cada operação de leitura em um item de dado x pela transação T_i no plano de execução S_1, se o valor lido x tiver sido escrito pela transação T_j, então a transação T_i deve também ler o valor de x produzido pela transação T_j na plano de execução S_2.
3. Para cada item de dado x, se a última operação de escrita em x foi executada pela transação T_i na plano de execução S_1, a mesma transação deve executar a escrita final do item de dado x na plano de execução S_2.

Como exemplo, considere as transações a seguir são serializáveis de leitura mas não serializáveis de conflito. As transações T_2 e T_3 não aderem à regra de restrição de escrita. Executam o que chamamos de **Escrita Cega**.

T_1	T_2	T_3
begin		
read(qtd_x)		
	begin	
	read(qtd_x)	
	write(qtd_x)	
	commit	
write(qtd_x)		
commit		
		begin
		read(qtd_x)
		write(qtd_x)
		commit

Transação 8: serialização de Visão

A serialização identifica planos de execuções que mantêm a consistência da base de dados desde que nenhuma das transações sobre a base de dados falhe. Uma alternativa é examinar a recuperação das transações em um plano de execução. Se uma transação falhar, a propriedade de atomicidade fará com que seja desfeito os efeitos da transação. Além disso, a propriedade de durabilidade atesta uma vez que uma transação se complete, suas alterações não podem ser desfeitas.

Considere novamente as transações definidas na **transação 8**. Em vez da operação *commit* ao final da transação T_1, um *rollback* é executado, de modo que os efeitos de T_1 sejam desfeitos. T_2 lê qtd_x alterada por T_1, e atualiza qtd_x, e confirma (*committed*) a alteração. Neste caso, T_2 precisa ser desfeita, uma vez que a alteração foi desfeita em T_1. Contudo, com a propriedade de durabilidade das transações, isto não será permitido. Este tipo de plano de execução é conhecido como **Plano de Execução Irrecuperável** e não deve ser permitido. Se torna então necessário de um **Plano de Execução Recuperável** onde, para cada par de transações T_i e T_j, se T_j ler um item de dado previamente escrito por T_i, então a operação *commit* de T_i precederá a operação *commit* de T_j.

Na prática, um SGBD não testa a serialização de um plano de execução. Isto é considerado impraticável, uma vez que o entrelaçamento das operações de transações concorrentes é determinado pelo sistema operacional. Em vez disso, a abordagem escolhida é usar protocolos que produzam planos de execuções serializáveis. São conhecidos como técnicas de controle de concorrência (bloqueios e *timestamp*) que serão discutidas nesta seção.

3.3. Técnicas de Controle de Concorrência

Permite a criação de plano de execuçãos serializáveis. Há duas técnicas que permitem que transações executem de forma segura e em paralelo; são elas, métodos de bloqueios (*locking*) e *de timestamp*. Os métodos são consideradas abordagens ***conservadoras*** ou ***pessimistas***, uma vez que fazem com que as transações se atrasem no caso de conflitos.

Bloquear

É um procedimento usado para controlar o acesso simultâneo dos dados. Quando a transação alcançar a base de dados, um bloqueio pode negar o acesso a outras transações e impedir resultados incorretos. É a aproximação mais usada para assegurar uma serialização.

Ao bloquear, uma transação obtém um travamento de um grupo de dados. Um bloqueio é um mecanismo que impede que uma outra transação modifique (e em alguns casos, leia) um grupo de dados. Pode ser um **Bloqueio de Leitura** (ou compartilhado), ou um **Bloqueio de Escrita** (ou exclusivo).

Se uma transação realizar um bloqueio de leitura, significa que pode ler o grupo de dados mas não escrevê-lo. Se uma transação realizar um bloqueio de escrita, significa que pode ler e escrever um grupo de dados. Desde que as operações lidas não se oponham, é permissível para mais de uma transação realizar simultaneamente bloqueios de leitura e de escrita para uma transação de acesso exclusivo a esse conjunto. Então uma transação que possui um bloqueio de escrita sobre um grupo de dados, nenhuma outra transação pode ler ou escrever neste grupo.

Os bloqueios são usados da seguinte maneira:

●Uma transação que necessita de um grupo de dados para leitura ou escrita deve primeiramente bloqueá-los.

●Se o grupo não estiver bloqueado por uma outra transação, este estará concedido.

●Se o grupo estiver atualmente bloqueado, o SGBD determina se o pedido é compatível com o bloqueio existente. Se um bloqueio de leitura for pedido no grupo que tem já um bloqueio de leitura o pedido é concedido. Se não, a transação deverá esperar até que o bloqueio seja liberado.

●Uma transação continua um bloqueio até que o libere explicitamente. Pode ser durante a execução ou na sua extremidade. Somente quando o bloqueio de escrita for liberado outra transação visualizará os efeitos da operação da escrita, isto é, as mudanças das operações da escrita não são visíveis a outros usuários ou aplicações.

É possível **promover** um bloqueio em qualquer sistema. Considere que a transação definida na **Transação 8**. Um novo plano de execução com bloqueios é mostrado a seguir:

T_1	T_2
begin	
write_lock(qtd_x)	
read(qtd_x)	
$qtd_x = qtd_x + 100$	
write(qtd_x)	
unlock(qtd_x)	
	begin
	write_lock(qtd_x)
	read(qtd_x)
	$qtd_x = qtd_x - 50$
	write(qtd_x)
	unlock(qtd_x)
	write_lock(qtd_y)
	read(qtd_y)
	$qtd_y = qtd_y - 50$
	write(qtd_y)
	unlock(qtd_y)
	commit
write_lock(qtd_y)	
read(qtd_y)	
$qtd_y = qtd_y - 100$	
write(qtd_y)	
unlock(qtd_y)	
commit	

Transação 9: plano de execução incorretamente travada

Uma transação realiza um bloqueio de leitura em um grupo de dados. Mais tarde este é promovido a um bloqueio de escrita. Isto faz com que a transação examine os dados primeiramente, e então, decida-se ou não a promoção do bloqueio. Caso a promoção não seja suportada, uma transação deve realizar bloqueios de escrita em todos os grupos de dados possíveis e atualizá-los durante sua execução. Isto reduz o potencial nível de simultaneidade do

sistema. Em alguns sistemas, esta reserva degrada um bloqueio a um bloqueio de leitura. Os bloqueios não garantem a serialização das programações.

Assumimos que a prioridade para executar qtd_x é 100 e qtd_y é 400. Executando T_1 primeiro antes de T_2 teremos como que o resultado de qtd_x é 150 e qtd_y é 250. Executando T_2 antes de T_1 teremos que o resultado de qtd_x é 150 e qtd_y é 450. Seguindo o plano de excução S, o resultado de qtd_x é 150 e qtd_y é 250. S **não** é um plano de execução serializado.

O problema deste exemplo é que este plano de execução libera os bloqueios que estão segurando as transações filhas associadas as operações de leitura/escrita que estão sendo executadas. Mas, a transação para si própria está bloqueando outros itens. Embora, permitindo as maiores concorrências, isto permite que as transações interfiram em outras transações resultando na perda do isolamento total e atomicidade.

Para garantir a serialização, um protocolo adicional de ser colocado na posição de travamento ou destravamento se necessário. Este protocolo é conhecido como ***two-phase locking (2PL)***.

No 2PL, a transação segue um protocolo que trava todas as operações precedentes e destrava a primeira operação de transação. Neste protocolo, a transação é dividida dentro de duas fases: **Crescimento** e **Encolhimento**. Na fase de crescimento, uma transação realiza todos os travamentos necessários mas não libera qualque um. Então, na fase de encolhimento, estes travamentos são liberados. Isto não é um requisito para obter travamentos simultâneos. Resumidamente as regras são:

- Uma transação deve obter um bloqueio de um item antes da operação deste. O bloqueio pode ser de leitura ou escrita, dependendo do tipo de acesso.
- Uma vez que uma transação libere um bloqueio, estão nunca poderá obter novos bloqueios.

Se promoções de bloqueios são permitidos, isto pode ter lugar somente durante a fase de crescimento, e deve requisitar esta transação a espera até que outra transação libere um bloqueio. De modo similar, minimizações pode ser feitas somente durante a fase de recolhimento. Devemos considerar que a 2PL resolve o **Problema da Perda de Alteração**. Veja a transação a seguir:

Tempo	**T_1**	**T_2**	**qtd_x**
t1		begin	150
t2	begin	write_lock(qtd_x)	150
t3	write_lock(qtd_x)	read(qtd_x)	150
t4	**WAIT**	qtd_x = qtd_x +100	150
t5	**WAIT**	write(qtd_x)	250
t6	**WAIT**	commit/unlock	250
t7	read(qtd_x)		250
t8	qtd_x = qtd_x - 10		250
t9	write(qtd_x)		240
t10	commit/unlock		240

Transação 10:Impedindo Problema da Alteração Perdida

Neste exemplo, a transação T2 pede um bloqueio de escrita em qtd_x. Uma vez concedido, pode adicionar 100 ao valor atual do qtd_x que é 150. Então, o novo valor (250) é escrito na base de dados. Quando a transação T1 tenta realizar uma alteração, solicita um bloqueio de escrita em qtd_x. Como foi travado por T2, T1 terá que esperar até que o bloqueio esteja liberado. Isto ocorre somente quando T2 completou a transação. Quando T1 realizar o bloqueio de escrita, lerá o valor novo de qtd_x (250). Assim, a consistência da base de dados é mantida.

Um outro exemplo mostra como a 2PL (***two-phase locking***) pode impedir o **Problema Não Comprometimento da Dependência**.

Tempo	T_1	T_2	qtd_x
t1		begin	150
t2		write_lock(qtd_x)	150
t3		read(qtd_x)	150
t4	begin	qtd_x = qtd_x +100	250
t5	write_lock(qtd_x)	write(qtd_x)	250
t6	**WAIT**	**rollback/unlock**	250
t7	read(qtd_x)		240
t8	qtd_x = qtd_x - 10		240
t9	write(qtd_x)		
t10	commit		

Transação 11: Impedindo o Problema do Não Comprometimento da dependência

A transação T2 pede um bloqueio de escrita em qtd_x. Pode então modificar o valor de qtd_x, e escrever na base de dados. Quando um *rollback* for executado, as modificações de T2 serão desfeitas e o valor de qtd_x na base de dados é retornado ao seu valor original. Quando a transação T1 inicia, e solicita um bloqueio de escrita em qtd_x. Entretanto, este não será concedido imediatamente já que ainda está bloqueado por T2. T1 deve esperar até que T2 libere através do *rollback*.

Tempo	T_1	T_2	qtd_x	qtd_y	qtd_z	sum
t1		begin	150	100	50	
t2	begin	sum = 0	150	100	50	0
t3	write_lock(qtd_x)	read_lock(qtdx)	150	100	50	0
t4	read(qtd_x)	**WAIT**	150	100	50	0
t5	qtd_x = qtd_x-10	**WAIT**	150	100	50	0
t6	write(qtd_x)	**WAIT**	140	100	50	0
t7	write_lock(qtd_z)	**WAIT**	140	100	50	0
t8	read(qtd_z)	**WAIT**	140	100	50	0
t9	qtd_z = qtd_z+10	**WAIT**	140	100	50	0
t10	write(qtd_z)	**WAIT**	140	100	60	0
t11	commit/unlock(qtdx, qtdz)	**WAIT**	140	100	60	0
t12		read(qtd_x)	140	100	60	0
t13		sum = sum + qtd_x	140	100	60	140
t14		read_lock(qtd_y)	140	100	60	140
t15		read(qtd_y)	140	100	60	140
t16		sum = sum + qtd_y	140	100	60	240
t17		read_lock(qtdz)	140	100	60	240
t18		read(qtd_z)	140	100	60	240
t19		sum = sum + qtd_z	140	100	60	300
t20		commit/unlock all read_locks	140	100	60	300

Transação 12: Impedindo o Problema de Inconsistencia da Análise

A 2PL pode também impedir o **Problema da Análise da Inconsistência**. Observe que a transação T1 precede o seu pedido com um bloqueio de escrita quando a transação T2 precede a sua leitura com um bloqueio de leitura. Quando T1 começa, solicita um bloqueio de escrita e estará concedido, e agora lata faz seus updates. O t2, na outra mão, não será concedido imediatamente o bloqueio. Tem que esperar o T1 para terminar e liberar os bloqueios. Assim, os valores que se usa encontrar a soma não estão sendo modificados por nenhuma outra transação. Os três exemplos acima mostram que toda a programação das transações que segue o protocolo travando bifásico está garantida para ser conflito serializable. Entretanto, quando o 2PL garantir a serialização, os problemas podem ocorrer com a interpretação de quando os bloqueios podem ser liberados como visto com um rollback sendo conectado em cascata como mostrado em figura 15.

Tempo	T_1	T_2	T_3
t1	begin		
t2	write_lock(qtd_x)		
t3	read(qtd_x)		
t4	read_lock(qtd_y)		
t5	read(qtd_y)		
t6	$qtd_x = qtd_x + qtd_y$		
t7	write(qtd_x)		
t8	unlock(qtd_x)	begin	
t9	:	write_lock(qtd_x)	
t10	:	read(qtd_x)	
t11	:	$qtd_x = qtd_x + 10$	
t12	:	write(qtd_x)	
t13	:	unlock(qtd_x)	
t14	:	:	
t15	rollback	:	
t16		:	begin
t17		:	read_lock(qtd_x)
t18		rollback	:
t19			rollback

Transação 13: Rollback Sendo conectado em cascata

A transação T1 realiza um bloqueio de escrita em qtd_x realiza uma modificação usando o valor de qtd_y que foi obtido pelo bloqueio de leitura e então atualiza a base de dados. A transação T2 realiza um bloqueio de escrita em qtd_x, e o modifica, então, envia à base de dados. A transação T3 realiza um bloqueio de leitura e lê qtd_x. Neste momento T1 falhou e procede um *rollback*. T2 é dependente de T1 quando tentou ler qtd_x e foi modificada por T1. T2 executa um *rollback*. De mesmo modo, T3 deve também executar um *rollback*.

Esta situação é conhecida como *rollback* em cascata. É indesejável porque processa uma quantidade significativa de trabalho. Seria útil se houvesse um protocolo que os impedisse. O único caminho para conseguir isto é através da liberação de todos os bloqueios até o fim do trabalho, como nos exemplos anteriores. Desta maneira, o problema mostrado aqui não ocorreria porque a transação T2 não obterá o bloqueio até que a transação T1 tenha terminado. Isto é conhecido como **2PL Rigoroso**. Um outro formulário de **2PL** é o **2PL Restrito**. Prende somente bloqueios de escrita até o fim da transação. A maioria dos SGBDs executa um destes dois **2PL**. Entretanto, um problema comum que se levante com esquemas trav-baseados é um beco sem saída. Um beco sem saída é uma situação em que nenhum progresso é possível porque dois ou os mais transações são cada um os bloqueios de espera prendidos pelos outros a ser liberados.

Time	T_1	T_2
t1	begin	
t2	write_lock(qtd_x)	begin
t3	read(qtd_x)	write_lock(qtd_y)
t4	$qtd_x = qtd_x$ -100	read(qtd_y)
t5	write(qtd_x)	$qtd_y = qtd_y + 10$
t6	write_lock(qtd_y)	write(qtd_y)
t7	**WAIT**	write_lock(qtd_x)
t8	**WAIT**	**WAIT**
t9	**WAIT**	**WAIT**
t10	:	**WAIT**
t11	:	:

Transação 14: deadlock

As transações T1 e T2 estão em *deadlock*, cada uma está esperando um bloqueio ser liberado. Em **tempo t2**, a transação T1 realiza um bloqueio de escrita em qtd_x, e modifica e grava um novo valor na base de dados. No **tempo t3**, a transação T2 realiza um bloqueio de escrita em qtd_y e tenta modificar e guardar o novo valor na base de dados. Quando a transação T1 no **tempo t6** faz um bloqueio de escrita na qtd_y, este não será dado porque está travado por T2. T1 então espera o bloqueio ser liberado. Quando T2 no **tempo t7** pede um bloqueio de escrita para qtd_x este não será dado pois esta travado por T1, então, espera T1 liberar o bloqueio.

No **tempo t8**, o *deadlock* inicia porque ambas estão esperando que os bloqueios serem liberados.

A única maneira quebrar um bloqueio é abortar uma ou mais transações. Isso envolve desfazer todas as mudanças realizadas pelas transações. Podemos decidir abortar T2 o que significa que todos os bloqueios de T2 serão liberado de modo que o T1 possa continuar com seu processar. Os *deadlocks* devem ser transparentes ao usuário. O SGBD deve reiniciar uma transação abortada.

Há duas técnicas gerais para segurar *deadlocks*.

1.**Prevenção de *deadlock***. O SGBD olha adiante para determinar se uma programação dada causar um beco sem saída, e nunca permite que um beco sem saída ocorra. Uma aproximação possível a esta deve requisitar transações usando os selos de tempo da transação (que serão discutidos na seção seguinte.)

2.**Detecção e recuperação de *deadlock***. O SGBD permite que os *deadlocks* ocorram entretanto reconheçam as ocorrências do *deadlocks*, e quebre-os. Segurado geralmente pela construção de uma esfera para o gráfico (WFG) que mostra dependências da transação, isto é, o T_i da transação é dependente de T_j, se a transação T_j prender o bloqueio em um item os dados que o T_i esteja esperando. O WFG é construído como segue:

- Cria um nó para cada transação
- Cria um caminho direto entre T_i->T_j
- Se T_i estiver esperando um bloqueio a ser liberado por T_j

Um *deadlock* existe somente se o WFG contém um ciclo. A seguinte figura demonstra o WFG de um *deadlock* que ocorreu na transação. O programa da detecção de *deadlock* gera o WFG em intervalos regulares e examina-o para um ciclo. A escolha do intervalo do tempo entre a execução do algoritmo é muito importante. Se o intervalo do tempo for demasiado pequeno, a detecção do *deadlock* adicionará consideráveis despesas no desempenho. Por outro lado, se o intervalo do tempo for longo, o *deadlock* não pode será detectado durante um longo período do tempo.

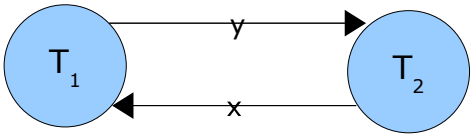

Figura 3: WFG da transação dois em figura 16

Desde que é mais fácil de detectar *deadlocks* e quebrá-los do que prevení-los, a maioria dos sistemas executa o método da deteção e da recuperação.

3.4. Métodos de Timestamp

Uma aproximação diferente garante a serialização no uso de transações com *timestamp* para requisitar a execução da transação para um equivalente plano de execução de série. Os métodos de *timestamp* para a concorrência não envolvem o uso de bloqueios. Conseqüentemente, nenhum *deadlock* ocorre. Os métodos baseados em bloqueio impedem a interferência deixando as transações em espera. O método de *timestamp* não realiza nenhuma espera. Para as transações em conflitos é realizado *rollback* e estas são reiniciadas.

Um ***timestamp*** é um identificador original criado pelo sistema que indica a época relativa de uma transação. Pode ser gerado usando um pulso de disparo do sistema indicando que a transação iniciou, ou por um contador lógico cada vez que uma transação nova começa.

Timestamp é um protocolo de controle simultâneo onde o objetivo fundamental é requisitar as transações globais de maneira que as transações mais velhas, transações com *timestamps* menores, possuam maior prioridade mesmo em caso de conflito. Se uma transação tentar ler ou escrever um grupo de dados, é permitida proseguir somente se a última modificação naquele conjunto for realizado por uma transação mais velha. Caso contrário, é reiniciada e dada um novo *timestamp*. Um novo *timestamp* é necessário para impedir que as transações sejam continuamente abortadas e reiniciaas. Se isto não fosse realizado, as transações com *timestamp* mais velho não poderiam ocorrer devido as transações mais novas que já estão ocorrendo.

Cada grupo de dados tem um ***read-timestamp*** que é o *timestamp* da última transação de leitura do item, e um ***read-write-timestamp*** que é o timestamp da última transação de escrita ou

atualização do item.

O ***timestamp ordering protocol*** é também conhecido como ***basic timestamp ordering***, e trabalha para uma transação T com ts(T):

1. T realizar uma leitura(x)
    - T solicita a leitura de um item (x) enquanto o valor está atualizado por uma transação mais nova, i.e., ts(T) < write_timestamp(x).

      Isto significa que T está tentando ler um valor de um item que foi atualizado por uma transação posterior. A transação está muito antiga para ler um valor prévio, e qualquer outro valor adquirido é provável ser incompatível com o valor atualizado do item de dados. Nesta situação, deve ser abortado T e deve ser reiniciado com um timestamp mais novo.
    - Por outro lado, ts(T) ≥ write_timestamp(x), e a operação de leitura pode ser procedida. Faça read_timestamp(x) = max(ts(T) < read_timestamp(x))
1. T realizar uma escrita(x)
    - T solicita a escrita de um item (x) enquanto o valor está sendo lido por uma transação mais nova, i.e., ts(T) < read_timestamp(x).

      Isto significa que uma transação posterior está usando um valor atual do item, e seria um erro atualizar isto agora. Então acontece quando uma transação estiver atrasada tentando escrever e uma transação mais jovem leu o valor antigo ou escreveu um novo. Neste caso, a única solução é realizar um rollback em T, e reiniciá-lo usando um timestamp posterior.
    - T solicita a escrita de um item (x) enquanto o valor está sendo atualizado por uma transação mais nova, i.e., ts(T) < write_timestamp(x).

      Isto significa que a transação T está tentando escrever um valor mais velho do item de dados x. Em T é feito um rollback e reiniciado usando um timestamp posterior.
    - Por outro lado, a operação de escrita é realizada. Faça write_timestamp(x) = ts(T).

Este esquema garante que transações são serializadas de conflito, e os resultados são equivalentes a um plano de execução consecutivo no qual as transações são executadas em ordem cronológica pelo timestamp. Porém, isto não garante planos de execução recuperáveis. Uma variação deste esquema é a **Regra de Escrita de Thomas** que rejeita operações de escrita obsoletas. Isto modifica a verificação para uma operação de escrita. São elas:

- T solicita a escrita de um item (x) enquanto o valor está sendo lido por uma transação mais nova, i.e., ts(T) < read_timestamp(x). Realizado um *rollback* em T e reiniciado um novo *timestamp*.
- T solicita a escrita de um item (x) enquanto o valor está sendo atualizado por uma transação mais nova, i.e., ts(T) < write_timestamp(x).

  Isto significa que uma transação posterior já alterada por um valor do item, e o valor que a mais velha transação está escrevendo deve ser um valor obsoleto do item. Neste caso, a operação de escrita pode ser seguramente ignorada. Isto às vezes é conhecido como o **Regra para Ignorar uma Escrita Obsoleto**, e permitir uma maior concorrência.
- Por outro lado, a operação de escrita é realizada. Faça write_timestamp(x) = ts(T).

Consideremos três transações ocorrendo concorrentemente conforme mostrado no diagrama abaixo. A segunda coluna (Op) mostra a ordem de execução das operações:

Tempo	Op	T_1	T_2	T_3
t1		begin		
t2	read(qtd_x)	read(qtd_x)		
t3	$qtd_x = qtd_x+10$	$qtd_x = qtd_x+10$		
t4	write(qtd_x)	write(qtd_x)	begin	
t5	read(qtd_y)		read(qtd_y)	
t6	$qtd_y = qtd_y+10$		$qtd_y = qtd_y+10$	begin
t7	read(qtd_y)			read(qtd_y)
t8	write(qtd_y)		write(qtd_y)	
t9	$qtd_y = qtd_y+10$			$qtd_y = qtd_y+10$
t10	write(qtd_y)			write(qtd_y)
t11	$qtd_z = 10$			$qtd_z = 10$
t12	write(qtd_z)			write(qtd_z)
t13	$qtd_z = 5$	$qtd_z = 5$		commit
t14	write(qtd_z)	write(qtd_z)	begin	
t15	read(qtd_y)	commit	read(qtd_y)	
t16	$qtd_y = qtd_y+10$		$qtd_y = qtd_y+10$	
t17	write(qtd_y)		write(qtd_y)	
t18			commit	

Transação 15: Concorrências

T_1 possui o *timestamp* $ts(T_1)=t_1$, T_2 tem $ts(T_2)=t_4$, e T_3 tem $ts(T_3)=t_6$ como $ts(T_1)<ts(T_2)<ts(T_3)$.

No tempo t_8, a operação de escrita de T_2 falha sobre o primeiro *timestamp* de escrita, i.e.:

$ts(T_2) < read\_timestampt(qtd_y)$

$t_4 < t_7$

Neste ponto, T_2 realiza um *rollback* e reinicia t_{14}.

No tempo t_{14}, a operação de escrita em qtd_z pode seguramente ignorar porque está sobrescrevendo o que já foi escrito pela transação T_3 no tempo t_7.

3.5. Técnicas Otimistas

Protocolos de Controle de Concorrência Otimista estão baseado na suposição que conflitos são raros. É mais eficiente que uma transação proceda sem impor qualquer demora para assegurar uma serialização. Neste esquema, se uma transação concluir (*commit*), uma verificação é feita para determinar se um conflito tiver ocorrido. Se houver um conflito, um *rollback* e feito e a transação é reiniciada. Considerando que a suposição raramente os conflitos acontecem, *rollbacks* serão raros. Reiniciar transações será aceitável para o desempenho do sistema. Há três fases de uma protocolo de controle de concorrência otimista:

1. **Fase de Leitura**. Começa imediatamente após o início da transação até o commit. A transação lê todos os itens de dados que precisa do banco e usa variáveis locais para segurar os itens. São aplicadas atualizações às cópias locais; não para o banco de dados.
2. **Fase de Validação**. Seguindo a fase de leitura. Nesta fase, são realizadas verificações para assegurar a serialização. Para as transações de leitura, são feitas verificações para assegurar que os valores dos dados são os valores atuais. Se forem atuais, a transação está comprometida. Caso contrário, a transação é abortada e é reiniciada. Para transação que possue atualizações, são feitos verificações para assegurar que a transação deixará o banco de dados em um estado consistente. Nesse caso, a transação está comprometida. Caso contrário, é abortado. Para passar a fase de validação, uma das seguintes opções deve ser verdadeira:
    - Toda a transação com timestamps novos deve ser terminada antes de uma transação mais nova possa ser iniciada.
    - Se uma transação iniciar antes que uma mais nova termine:
        - O conjunto de dados escrito pela transação mais nova não é o prioritariamente lido pela transação atual; e
        - A transação mais nova completa sua fase de escrita antes que a transação atual

entre na fase de validação.

1. **Fase de Escrita**. Seguida de uma verificação com sucesso da fase de validação. São aplicadas as atualizações feitas à cópia local ao banco de dados.

3.6. Granularidade dos Itens de Dados

Todos os protocolos de controle de concorrência discutidos assumem um acesso e modicação dos dados está baseado em uma única coluna. Em geral, podemos recorrer aos dados como um item de dados, e pode ser definido por sua granularidade. Granularidade é um lado dos itens de dados escolhido como a unidade de proteção por um protocolo de controle de concorrências. Fina granularidade é obtida de itens pequenos e grossa granularidade é obtida de grandes itens. Granularidade de itens de dados pode ser:

- Base inteira.
- um arquivo.
- uma página (uma seção de armazenamento físico).
- um registro.
- um valor de um campo.

Há intercâmbios que podem ser considerados escolhendo o item de dados. O intercâmbio discutido estaria no contexto de bloquear. Embora, podem ser feitos argumentos semelhantes para outras técnicas de controle de concorrência. O princípio que aplica é:

> O mais grosso item de dados classifica segundo o tamanho, o mais baixo grau de concurrência permitido. Quanto mais fino o item de dados é, mais a informação é bloquada e será armazenada.

O melhor tamanho de um item de dados dependeria da natureza das transações. Se uma transação acessa e atualiza um número pequeno de registros, é aconselhável para ter a granularidade do item de dados ao nível de registro. Por outro lado, se uma transação acessa e modifica muitos registros de dados, seria melhor para definir a granularidade a nível da tabela.

O granularidade dos itens de dados podem ser apresentadas como uma estrutura hierárquica onde cada nó representa itens de dados de tamanhos diferentes como mostrado na Figura 4. O nó de raiz representa o banco de dados inteiro. O próximo nível de nós representa tabelas. O próximo nível a página. O próximo nível o registro. E finalmente, o próximo nível representa os campos ou colunas. O princípio da hierarquia é:

> Se um nó é bloqueado, todos os seus descendentes são bloqueados.

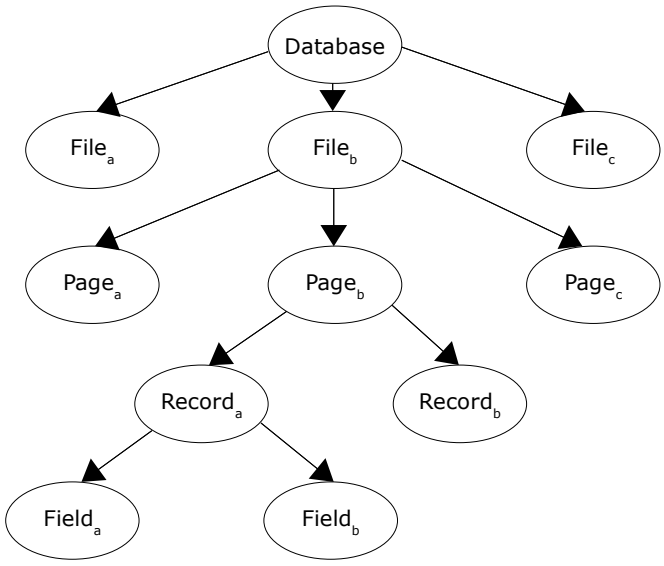

Figura 4: Hierarquia de Granularidade dos Itens de Dados

Se uma transação bloqueia $Page_b$, todos os registros também serão bloqueados, seus campos ou colunas. Se uma outra requisição de transação de um bloqueio incompatível no mesmo nó, o sistema conhecerá facilmente que este bloqueio não pode ser concedido.

De forma semelhante, se outra transação solicita um bloqueio em qualquer dos descendentes de $Page_b$ como $Record_a$, o sistema verifica o caminho da hierarquia da raiz até o nó da requisição para determinar se quaisquer de seus antepassados será bloqueado antes de permitir ou não o bloqueio. Neste caso, desde que $Page_b$ for bloqueado, o pedido de bloqueio para $Record_a$ será negado.

Adicionalmente, uma transação pode pedir um bloqueio em um nó no qual um ou mais de seus descendentes serão bloqueados. Neste cenário, a estratégia é usar outro tipo de bloqueio chamado de **Bloqueio de Intenção**. Um bloqueio de intenção bloqueia os antepassados de um nó, i.e., quando um nó é bloqueado, todos seus antepassados são determinados por um bloqueio de intenção de intenção. Visualizando a Figura 4, quando $Record_a$ for fechado, e alguma transação pedir um bloqueio em $File_b$, a presença de um bloqueio de intenção em $File_b$ indica que um descendente foi fechado.

Um bloqueio de intenção pode ser compartilhado (read) ou exclusivo (write).

1. Uma bloqueio de intenção compartilhada (IS) conflita com um bloqueio exclusivo.
2. Uma bloqueio de intenção exclusivo (IX) conflita com com ambos bloqueios
3. Uma transação pode segurar um compartilhamento e um bloqueio de intenção exclusiva (SIX) é logicamente equivalente a segurar ambos um compartilhamento e bloqueio de IX.
4. Um bloqueio de conflito SIX com qualquer bloqueio de conflito pode ser compartilhado com outro bloqueio IX.

A matrix de compatibilidade é mostrada na Tabela 1.

Locks	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

✓ - compatível; ✗- incompatível

Tabela 1: Matrix de Compatibilidade de Bloqueio

Para assegurar a serializacação com os níveis de bloqueio, um protocolo 2PL é usado conforme as seguintes regras:

- Nenhum bloqueio pode ser concedido uma vez que um nó foi desbloqueado.
- Nenhum nó pode ser bloqueado até que seu pai seja fechado por uma bloqueio intencional.
- Não existe modo para desbloquear até que todos seus descendentes são desbloqueados.

Não existe modo de desbloquear até que todos seus descendentes são desbloqueados.

Usando estas regras, bloqueios podem ser aplicados na raiz, usando bloqueios de intenção até que o nó possa requerer um bloqueio de leitura ou escrita. Bloqueios são liberados de baixo para cima. Paralisações completas podem ainda acontecer, e deve ser controlado conforme discutido.

4. Recuperação no Banco de Dados

Database Recovery is the process of restoring the database to a correct state in the event of a failure. A SGBD must provide database recovery services in an event that the system encounters a failure. In this context, reliability of a SGBD refers to both the resilience of the system to various types of failure and its capability to recover from them. To gain a better understanding of the potential problems we may encounter in providing a reliable system, we start at examining the need for recovery, and the types of failure that can occur in a database environment.

Recuperação de banco de dados é o processo que restabelece o banco de dados a um estado correto no caso de uma falha. Um SGBD deve providenciar um serviço de recuperação do banco em um evento para localizar as falhas. Neste contexto, um SGBD de confiança recorre a **elasticidade** do sistema para vários tipos de falhas e a sua **capacidade** de se recuperar. Obtendo um entendimento melhor dos problemas em potenciais, podemos encontrar meios para um sistema seguro, começaremos então examinando a necessidade de recuperação, e os tipos de falhas que podem acontecer em um ambiente de banco de dados.

4.1. Porque necessitamos de Recuperação?

Há muitos tipos diferentes de falhas que podem afetar o processo do banco de dados. Cada um das quais precisa ser corrigida de maneira diferente. Algumas das causas de falhas são:

- *System Crashes*. Se refere a erro de hardware ou software que afetam a perda de dados na memória principal.
- *Media Failure*. Se refere aos nossos dispositivos de armazenamento secundários como discos magnéticos, discos ópticos, e fitas. Pode ser problemas no *head* ou mídia ilegíveis que resultam na perda de partes de dados em armazenamento secundário.
- *Application Software Errors*. Se refere a erros lógicos no programa que acessa e atualiza os bancos de dados que podem fazer uma ou mais transações falhar.
- *Natural Physical Disasters*. Se refere a fogo, inundações, terremotos, ou deficiências de energia.
- *Carelessness or Unintentional Destruction*. Se refere a ações, intencionais ou não, dos usuários ou operadores do sistema.

Seja qual for a causa da falha, dois principais efeitos são considerados para a recuperação do banco de dados.

1. Perda dos dados na memória principal incluindo os bancos temporários
2. Perda dos dados copiados na base de dados

A seção seguinte discute os conceitos e as técnicas que podem minimizar estes efeitos permitindo a recuperação de falha.

4.2. Relacionamento entre Transações e Recuperação

O componente que controla a recuperação de banco de dados é o **Gerente de Recuperação** como foi visto na Figura 1. Sua responsabilidade é assegurar duas propriedades das transações, **Atomicidade** e **Durabilidade**. Deve assegurar que, ou em recuperação de uma falha, são registrados todos os efeitos de uma determinada transação permanentemente no banco de dados ou nenhum deles são. A situação é complicada pelo fato de que uma transação pode gravar na memória principal mas seu efeito não é permanente e simplesmente foi escrito no banco de dados contudo alcançou o armazenamento secundário. Considere as seguintes transações:

```
read(item_code=x and store_no=y, qtd)
qtd = qtd + additional_stock
write(item_code=x and store_no=y, qtd)
```

O SGBD leva a cabo os passos seguintes para processar a leitura em qtd_x.

1. Encontra o endereço do bloco no disco que contém o registro de qtd_x utilizando sua chave primária.
2. Transfere esse bloco do disco em um buffer do banco de dados na memória principal.
3. Copia o dado qtd_x do buffer para a variável qtd_x.

Para a operação de escrita de qtd_x, as seguintes etapas são realizadas:

1. Encontra o endereço do bloco no disco que contém o registro de qtd_x utilizando sua chave primária.
2. Transfere esse bloco do disco em um buffer do banco de dados na memória principal.
3. Copia o valor da variável qtd_x para o buffer.
4. Grava em disco o conteúdo do buffer.

A título de revisão, buffers de banco de dados são porções, ou áreas, na memória principal de onde (ou para onde) os dados são transferidos para o (ou a partir do) armazenamento secundário. Somente a partir do momento em que os buffers são descarregados no armazenamento secundário consideramos a transação como permanente. O descarregamento de buffers pode ser disparado de qualquer uma das formas abaixo indicadas:

- Por um comando específico tal como um "transaction commit". Também conhecido como ***force-writing***.
- Automaticamente, quando o buffer estiver cheio.

Devido a possibilidade de falha entre a escrita no buffer e sua gravação no armazenamento secundário, o gerenciador de recuperação deve identificar o status da transação que executava a operação de gravação no momento da falha. Caso a transação tenha sido finalizada (committed), o gerenciador de recuperação, para garantir durabilidade, deve executar novamente a atualização da transação no banco. Conhecido como ***rollforward***. Por outro lado, caso a transação não tenha sido finalizada (committed), o gerenciador de recuperação deve desfazer, ou ***rollback***, quaisquer conseqüências dessa transação no banco para garantir atomicidade.

Se somente uma transação for desfeita, têm-se um ***partial undo***, disparado pelo scheduler dentro do evento em que a transação é desfeita e reiniciada devido ao protocolo de controle de concorrência. Uma transação também pode ser abortada unilateralmente, como no caso em que o usuário ou uma aplicação dispara uma exceção. Quando todas transações ativas são desfeitas, temos um ***global undo***.

4.3. Mecanismo de Recuperação

O SGBD deve oferecer o seguinte mecanismo para auxiliar a recuperação do banco

1. *Backup Mechanism*. Gera cópias de backup do banco. As cópias de backup e o arquivo de log (discutido a seguir) são gerados em intervalos regulares sem a necessidade de paralisar o sistema. As cópias de backup são utilizadas para restaurar completamente o banco em caso de ter sido danificado ou destruído. Existem dois tipos de cópias de backup: completa e incremental. O backup completo, ***full backup***, é o backup de toda a base enquanto que o backup incremental consiste somente das modificações feitas deste o último backup completo ou incremental. O backup é armazenado offline e, geralmente, em fitas magnéticas.
2. *Logging Mechanisms*. Um arquivo que registra o estado atual das transações e das mudanças no banco. Também conhecido como *journal*. Pode conter as seguintes informações:
    - Registros de Transação que consistem de:
        - Identificador da Transação: identifica a transação unicamente
        - Tipo de Registro: pode ser *transaction start*, *insert*, *update*, *delete* etc
        - Identificador do item de dado afetado pela ação no banco

- Valor anterior do dado antes da alteração
- Novo valor do dado após a alteração
- Informação de Gerenciamento de Log: pode ser um ponteiro para o registro de log anterior ou seguinte

Um trecho de um arquivo de log é exibido na Tabela 2 e mostra três transações executando concorrentemente, T_1, T_2 and T_3. PPtr e NPtr representam ponteiros para os registros de log seguintes e anteriores para cada transação.

TID	Tempo	Operação	Objetivo	Imagem Anterior	Imagem Posterior	PPrts	NPtrs
T_1	9:15	START				0	2
T_1	9:30	UPDATE	INVENTORY qtd_x	100	150	1	8
T_2	9:33	START				0	4
T_2	9:40	INSERT	INVENTORY qtd_y		300	3	5
T_2	9:42	DELETE	ITEM w	w		4	6
T_2	9:45	UPDATE	STORE 40	Ohoi	Ohio	5	9
T_3	9:57	START				0	11
T_1	9:57	COMMIT				2	0
	10:01	CHECKPOINT	T1, T2				
T_2	10:03	COMMIT				6	0
T_3	10:05	INSERT	ITEM g		g	7	11
T_3	10:10	COMMIT				11	0
T_3	10:12	INSERT	STORE 50		50	12	0

Tabela 2: Segmento de um arquivo de Log

Dada a importância de arquivos de log, é necessário que se tenha mais de uma cópia de modo que se a primeira cópia apresentar problema, a segunda ou terceira cópia possa ser utilizada.

É altamente recomendado que os arquivos de log sejam gravados em um disco diferente do que armazena o banco.

- *Checkpoint Records*. Utilizado pelo mecanismo de *checkpoint*.

1. *Checkpoint Mechanism*. Permite que atualizações no banco sejam permanentes. Um *checkpoint* é o ponto de sincronização entre o banco e o log da transação. Indica o ponto em que se dispara o comando para que todos os buffers sejam gravados no armazenamento secundário. Para a recuperação da base, auxilia a determinar o quanto se deve retornar no log para a restauração. É agendada em intervalos predefinidos que envolvem as seguintes operações:
   - Gravação de todos os registros da memória principal no armazenamento secundário
   - Gravação de todos os blocos modificados na memória principal no armazenamento secundário
   - Gravação de um registro de *checkpoint* no arquivo de log. Este registro contém os identificadores de todas as transações que estão ativas no momento do *checkpoint*.
2. Gerenciamento de Recuperação. Permite que o sistema recupere a base para um estado consistente após uma falha.

4.4. Técnicas de Recuperação

O procedimento específico de recuperação a ser utilizado depende da extensão do dano na base. Nesta seção, dois procedimentos serão apresentados.

1. Se um banco de dados foi muito danificado tal como dano ao disco por quebra da cabeça de leitura, deve-se, necessariamente, restaurar o banco a partir do último backup completo seguido das atualizações gravadas no arquivo de log de transações finalizadas.
2. Caso o banco de dados não tenha sido danificado fisicamente mas se tornou inconsistente devido a problema no sistema durante a execução de transações, deve-se desfazer as alterações causadas pela inconsistência. Também é possível reiniciar transações para assegurar que as alterações executadas tenham chegado ao armazenamento secundário. Nesse cenário, a restauração a partir do backup completo não é necessária; deve-se utilizar somente o arquivo de log que contém as informações anteriores e seguintes às transações.

Na seção seguinte, veremos técnicas de recuperação utilizadas onde a base esteja em estado inconsistente. Existem duas: alteração atrasada (*deferred update*) e alteração imediata (*immediate update*)

Recuperação com Deffered Update

Nesta técnica as atualizações são gravadas na base somente após o commit da transação. Se a transação falhar antes da gravação das alterações, não há nada a ser desfeito. Entretanto, pode ser necessário refazer as alterações da transação finalizada pois há a possibilidade de que não tenha gravado suas alterações na base. O arquivo de log é necessário para proteção contra falhas. A seguir, as etapas executadas pelo SGBD para efetuar a gravação da uma transação utilizando os arquivos de log:

1. Quando uma transação inicia, grava um registro de START no arquivo de log ou journal.
2. Quando qualquer operação de gravação é realizada, insere um registro no arquivo de log contendo dados da transação, conforme mostrado na Tabela 2. Os dados anteriores não são incluídos. Neste ponto, a operação de gravação não é executada nem em buffer nem na própria base.
3. Quando uma transação está prestes a finalizar, grava um registro de log COMMIT e grava todos os registros de log no disco. A seguir, finaliza a transação. Utiliza os registros de log para efetuar as alterações no banco.
4. Caso uma transação aborte, ignora todos os registros de log para a transação e não executa as operações de gravação.

Note que os registros de log são escritos no disco antes que a transação execute a real finalização. Isso é feito para que quando ocorrer uma falha de sistema durante uma atualização, o registro de log sobreviva e as alterações possa ser efetivada posteriormente. Quando uma falha ocorre o arquivo de log é utilizado para identificar as transações que estavam em progresso no momento da falha. Para recuperar-se de uma falha, o SGBD executa o seguinte:

1. Qualquer transação com registros de START e COMMIT devem ser reiniciadas e refeitas. O procedimento é gravar na base com os valores posteriores à alteração no arquivo de log na mesma seqüência em que foram gravados no log. Isso garante que os dados serão alterados de qualquer forma.
2. Qualquer transação com registros de START e ABORT deve ser ignorada.

Recuperação com Immediate Update

Nesta técnica as alterações são aplicadas na base à medida em que chegam ao commit. De modo semelhante à técnica anterior, o arquivo de log é necessário para proteção contra falhas. A seguir, as etapas executadas pelo SGBD para efetuar a gravação de uma transação nos arquivos de log:

1. Quando uma transação inicia, grava um registro de START no arquivo de log.
2. Quando uma operação de gravação é executada, o registro contendo o dado é gravado no arquivo de log.
3. Uma vez que o registro de log é gravado, executa a operação de gravação nos buffers do banco.
4. A alteração no banco é gravada quando os buffers são descarregados no armazenamento

secundário.

5. Quando a transação finaliza, um COMMIT é gravado no arquivo de log.

É necessário que arquivos de log sejam gravados antes que as operações de gravação correspondentes no banco sejam executadas. Caso as alterações fossem executadas antes do log e uma folha ocorresse, o gerenciador de recuperação não teria como desfazer ou refazer a operação. A gravação antecipada do log é conhecida como ***write-ahead log protocol***. O gerenciador de recuperação pode operar seguramente quando não há registro de COMMIT para uma transação o que significa que ainda estava ativa no momento da falha e, portanto, precisa ser desfeita.

Caso a transação aborte, o arquivo de log pode ser utilizado para desfazê-la pois contém todas as informações anteriores às operações de gravação. As gravações são desfeitas em ordem reversa. Independente das gravações terem sido executadas, a posse dos dados anteriores garante que a base será restaurada ao seu estado anterior ao do início da transação.

Se o sistema falhar, a recuperação envolve a utilização do log para desfazer ou refazer transações. A recuperação é executada como segue:

1. Qualquer transação com registros de START e COMMIT serão refeitas com as informações seguintes à alteração.
2. Qualquer transação com registro de START mas não de COMMIT serão desfeitas com as informação anteriores à alteração. As operações de desfazer são executadas em ordem reversa da qual foram gravadas no arquivo de log.

5. Suporte a Transação com JDBC

As operações de transação de JavaDB são executadas através de procedimentos JDBC e não por comandos SQL. Em JDBC, uma transação é um conjunto de um ou mais comandos SQL que formam uma unidade de trabalho lógica que podem tanto ser finalizadas quanto desfeitas. Todas as declarações na transação são atômicas, e a definição da transação ajuda na recuperação da base em caso de falha.

Uma transação é associada com um único objeto de conexão com a base que organiza e envia todos comandos SQL para um mesmo objeto `Connection` desde o último commit ou rollback. Não pode ser aplicada a várias conexões ou bancos de dados. Para gerenciar uma transação, utilizamos os seguintes métodos do objeto `Connection`:

1. `setAutoCommit()`. Este método ajusta o *auto-commit* da conexão. *Auto-commit* significa que o banco irá persistir automaticamente as alterações na base após a execução de um comando SQL. Aceita valores *boolean.* Se for *true*, indica que o *auto-commit* está ativo, caso contrário, que está desativado e *auto-commit* é *false*. Quando *auto-commit* está desativado, os métodos `commit(` e `rollback()` podem ser utilizados para gerenciar transações.
2. `commit()`. Faz com que todas as alterações desde o último *commit/rollback* tenham sido executados e libera o lock da base atualmente de posse desse objeto `Connection`.
3. método rollback (). Desfaz todas as alterações feitas na transação atual e libera qualquer trava de banco de dados atualmente detidos pelo objeto Connection.

Algumas aplicações podem preferir trabalhar com o auto-commit. Outras poderiam desejar que ele seja desligado. O programador deve estar ciente das implicações de utilizar qualquer modelo. Isto é necessário para o seguinte:

1. *Em Cursores.* Auto-commit não pode ser utilizado se posicionado em atualizar ou suprimir WHERE CURRENT OF-clause é usado com a opção de fechar cursor é verdade ou ligado. Um cursor declarado a ser realizado em todo o registro pode executar várias atualizações e emite vários registros antes de fechar o cursor. No entanto, o cursor deve ser reposicionado antes de qualquer declaração após o registro. Se esta é a tentativa de auto-commit, um erro é gerado. Cursores são discutidos na próxima seção.
2. *em Database-lateral JDBC Procedimentos e funções.* Procedimentos e funções dentro de comandos SQL não podem ser executadas se esses processos e funções desenvolverem um registro ou reversão, pois registros são implicitamente realizados após um comando SQL em auto-commit. Procedimentos e funções que usam conexões aninhadas não estão autorizados a transformar auto-commit ligado ou desligado.
3. em Tabela de nível de bloqueio e SERIALIZABLE Nível de isolamento. Quando uma aplicação usa tabela de nível de bloqueio e isolamento de nível serial , todos os comandos que acessam tabelas travadas, pelo menos compartilharam tabelas de bloqueio. Bloqueios compartilhados impedem que outras transações de atualização de dados acessem a tabela. A operação realiza um bloqueio em uma tabela até que a transação seja encerrada.

A Tabela 3 mostra um resumo do comportamento com a aplicação Auto-commit ligado ou desligado

Conceitos	Auto-commit ligado	Auto-commit desligado
Transações	Cada comando é uma operação separada.	Os métodos commit() ou rollback() encerram uma transação.
Comandos de Database-side JDBC com comandos SQL e conexões aninhadas	Cada comando SQL-é considerado como uma operação, ou seja, ele fecha automaticamente após cada execução de comando.	commit() e rollback() define quando a operação é encerrada.
Updatable Cursors	Não funciona.	Funciona.
Múltiplas conexões acessando os mesmos dados	Funciona.	Funciona. No entanto, tem menor concorrencia quando utiliza aplicações de nível de isolamento serial e tabela de nível de bloqueio.
Updatable ResultSets	Funciona	Funciona. No entanto, não é exigido pelo programa JDBC.

Tabela 3: Aplicações em comandos com Auto-commit ligado ou desligado

Serviços de codificação de operação basicamente incluem início da operação, executando os comandos SQL que compõem a operação, e que executa um comando com sucesso total de cada comando SQL ou rollback da operação como um todo se uma das instruções SQL falhar.

Sempre que uma nova conexão é aberta, a operação de modo auto-commit está ligado. Em modo auto-commit, cada comando SQL é executado como uma única operação que se remete imediatamente para o banco de dados. Para executar múltiplas instruções SQL como parte de uma única operação, o auto-commit precisa de ser desativado.

Por padrão, operação da DML INSERT, UPDATE, DELETE ou comandos em programas JDBC é feito automaticamente, porque o auto-commit está definido ligado por padrão. No entanto, se você optar por definir o modo auto-commit desligado, você pode fazê-lo, chamando a setAutoCommit () do objeto Connection como segue:

```
theConnection.setAutoCommit(false);
```

A linha de código acima deve aparecer imediatamente após a conexão tiver sido estabelecida como mostrado no texto 1. Uma vez que o modo auto-commit é desligado, um explícito COMMIT ou rollback deveria ser feito para cometer quaisquer alterações não salvas banco de dados. COMMIT ou rollback pode ser feito chamando commit () ou rollback () os métodos do objeto `Connection` como mostrado abaixo.

```
theConnection.commit();
```

ou

```
theConnection.rollback();
```

Considere o código 1 que define as operações JDBC para a classe *TransactionExample* é a execução da operação definida na ordem em que são colocadas em um banco de dados. Nessa operação, devemos subtrair a quantidade de um determinado item no inventário quando é colocado um fim nesse item.

Duas tabelas adicionais são criadas para realizar a ordem dos itens, nomeadas como `orders` e `order_details`.

- A tabela *orders* contém cabeçalho que contém o número de ordem (orderNbr), data fim (orderDate) e o número dos clientes (customerID).
- A tabela *order_details* contém os itens ordenados por uma ordem especial de vendas. Possui informações como o número de ordem (orderNbr), número entrada do item (lineNbr), lugar em que os itens foram ordenados (`store`), código do item(item), e da quantidade ordenada (`qty`).

Codigo 1: Transações no método main

```
public static void main(String[] args) throws SQLException {
  int ret_code;
  String url = "jdbc:derby://localhost:1527/organicshop";
```

```
String username = "nbuser";
String password = "nbuser";
String input = null;
char ans = 'Y';
Connection theConnection = null;
try {
  // Get a connection.  Implicitly, will start your connection.
  Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
  theConnection = DriverManager.getConnection(url, username, password);
  //Disable auto-commit mode
  theConnection.setAutoCommit(false);
  // Creating the Order
  int orderNbr, customerId;
  String orderDate;
  int lineNbr, store, item, qty;
  while (ans == 'Y'){
    JOptionPane.showMessageDialog(null, "Entering Orders");
    input = JOptionPane.showInputDialog("Enter the order number: ");
    orderNbr = Integer.parseInt(input);
    orderDate = JOptionPane.showInputDialog("Enter order date: ");
    input = JOptionPane.showInputDialog("Enter customer number: ");
    customerId = Integer.parseInt(input);
    input = JOptionPane.showInputDialog("Enter store number: ");
    store = Integer.parseInt(input);
    //Inserts the Order Header
    insertOrder(theConnection, orderNbr, orderDate, customerId);
    char ansToMore = 'Y';
    while(ansToMore == 'Y'){
      input = JoptionPane.showInputDialog(
        "Enter line number of order detail: ");
      lineNbr = Integer.parseInt(input);
      input = JOptionPane.showInputDialog("Enter the item code: ");
      item = Integer.parseInt(input);
      input = JoptionPane.showInputDialog("Enter the qtd ordered: ");
      qty = Integer.parseInt(input);
      //Inserts the Details of the Order
      insertOrderDetail(theConnection,orderNbr,lineNbr,store,item,qty);
      updateInventory(theConnection, item, store, qty);
      input = JOptionPane.showInputDialog("Enter another item [Y/N]");
      if (input.length()>0){
        ansToMore = input.charAt(0);
      } else {
        ansToMore = 'Y';
      }
    }
    input = JOptionPane.showInputDialog("Enter another order [Y/N]: ");
    if (input.length()>0){
      ans = input.charAt(0);
    } else {
      ans = 'Y';
    }
  }
  // If all goes well, we need to commit the transaction.
  theConnection.commit();
  theConnection.close();
} catch(ClassNotFoundException cnfx){
  System.err.println("Organic Shop: Cannot Load Driver");
  cnfx.printStackTrace();
  System.exit(1);
} catch(SQLException e) {
  // Rollback all the changes so as to undo
  // the effect of both INSERT and UPDATE
  System.err.println("Organic Shop: Transaction is not successful." +
    "Rolling back to a consistent state.");
  theConnection.rollback();
  ret_code = e.getErrorCode();
```

```
        System.err.println(ret_code + e.getMessage()); theConnection.close();
      } // end of try-catch-block
    } // end of main
```

O código 2 define uma operação que consistirá de um comando de inserção em `orders` e `order_details` e comando de atualização em `inventory` no qual são definidos três métodos particulares:

1. Método `insertOrder()` que insere um registro na tabela `orders`.
2. Método `insertOrderDetail()` que insere um registro na tabela `order_details`.
3. Método `updateInventory()` que atualiza a quantidade do item, ou seja, subtrai a quantidade ordenada a partir da atual quantidade de estoque adequado de registro.

Codigo 2: Transações em métodos particulares

```
private static void insertOrder (Connection theConnection, int orderNbr,
    String orderDate, int customerID) throws SQLException {
  String insertStmt =  "INSERT INTO orders VALUES( " + orderNbr +
    " , DATE('";
  insertStmt = insertStmt + orderDate + "'), " + customerID + ")";
  Statement theStatement = theConnection.createStatement();
  theStatement.executeUpdate(insertStmt);
  System.out.println("Insert Order is successful.");
  theStatement.close();
}
private static void insertOrderDetail(Connection theConnection, int orderNbr,
    int lineNbr, int store, int item, int qty) throws SQLException{
  String insertStmt = "INSERT INTO order_details VALUES (?,?,?,?,?)";
  PreparedStatement pstmt = theConnection.prepareStatement(insertStmt);
  pstmt.setInt(1, orderNbr);
  pstmt.setInt(2, lineNbr);
  pstmt.setInt(3, item);
  pstmt.setInt(4, store);
  pstmt.setInt(5, qty);
  pstmt.execute();
  System.out.println("Insert Order Detail successful.");
  pstmt.close();
}
private static void updateInventory(Connection theConnection, int item,
    int store,int qty) throws SQLException{
  String updateSQL1 = "UPDATE inventory SET qtd = qtd - ? " +
    "WHERE store_no = ? AND item_code = ?";
  PreparedStatement pstmt = theConnection.prepareStatement(updateSQL1);
  pstmt.setInt(1, qty);
  pstmt.setInt(2, store);
  pstmt.setInt(3, item);
  pstmt.execute();
  pstmt.close();
  System.out.println("Update of Inventory is successful.");
}
```

Conforme mostrado, a chamada para as funções particulares faz parte da tentativa de bloqueio. Se a execução dos comandos DML são bem sucedidas, o método `theConnection.commit()`é chamado a indicar escrevendo as alterações no banco de dados. No entanto, se ocorrer um erro, o comando DML irá criar uma SQLException e será pego pelo bloco de captura contendo SQLException. Neste bloco, `theConnection.rollback()` é chamado para desfazer as alterações que ocorreram antes do erro ou falha do banco de dados.

Um exemplo de uma mensagem quando um SQLException é suscitada é mostrado na figura a seguir:

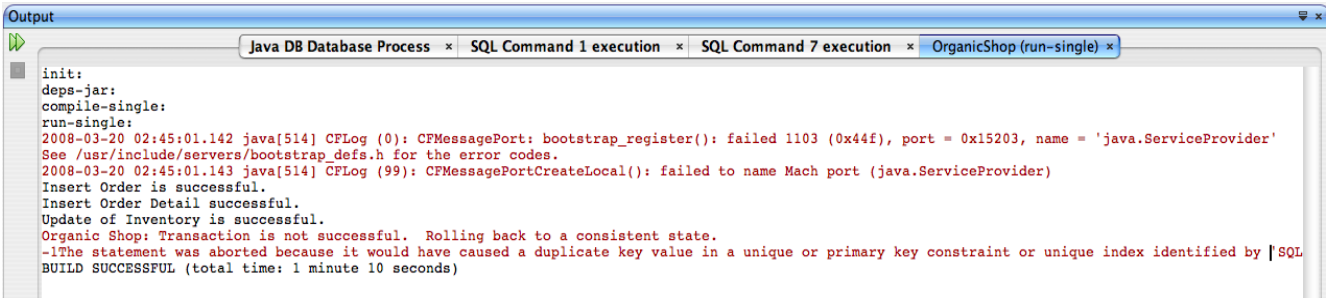

Figura 5: Rollback Message Example

Neste exemplo, um erro ocorreu porque ao tentar introduzir um detalhe (order_details) causou um constrangimento na chave primária. Inserções e atualizações bem sucedidas são desfeitas no ponto em que é chamada uma SQLException.

1. COMMIT ou ROLLBACK é feito para uma declaração e não para um comando individual DML.
2. É necessário encerrar ou repetir a primeira conexão andes de fechá-la.
3. DDL é um procedimento em Java, DB. Isto é, DDL NÃO compromete implicitamente a operação. Por exemplo, se algumas linhas forem inseridas e, em seguida, for criada uma tabela e, em seguida for revertida a operação, as linhas inseridas serão retiradas, juntamente com a tabela criada.
4. Desabilitar auto-commit melhora o desempenho em termos de tempo e de processamento como um COMMIT não precisa de ser emitido para cada comando SQL que afetem a base de dados.

5.1. Controle de Protocolo simultâneo: níveis de bloqueios e isolamento

Níveis de isolamento

A visibilidade das alterações feitas no banco de dados para o resto do sistema de aplicação é denominado como o nível de isolamento, ou seja, em um sistema multi-usuário, quando as alterações realizadas por um usuário se tornam visíveis para os utilizadores restantes? Transações podem operar em diversos níveis isoladamente. Defini-lo para uma conexão permite um usuário especificar a forma como severamente o usuário da operação deve ser isolado de outras operações.

JDBC fornece uma aplicação com uma maneira de especificar um nível de isolamento através do objeto Connection. JDBC e JavaDB prevêem quatro níveis de operação isolada, e seu mapeamento é apresentado na Tabela 2.

Nível de isolamento para JDBC (do objeto Connection)	Nível de isolamento para ANSI SQL
TRANSACTION_READ_UNCOMMITTED	UR, DIRTY READ, READ UNCOMMITTED
TRANSACTION_READ_COMMITTED	CS, CURSOR STABILITY, READ COMMITTED
TRANSACTION_REPEATABLE_READ	RS
TRANSACTION_SERIALIZABLE	RR, REPEATABLE READ, SERIALIZABLE

Tabela 4: JDBC e JavaDB Mapeamento do nível de isolamento

O isolamento de níveis evita determinados tipos de anomalias de operação que são descritas na Tabela 3.

Anomalia operacional	Explicação
Má leitura	Uma má leitura acontece quando uma transação lê dados que estão sendo modificados por uma outra operação que ainda não tenha acontecido. Exemplo:

Anomalia operacional	Explicação

Leitura não repetida	Operação A começa.
	`UPDATE inventory SET qtd = qtd – 50 WHERE store_no = 10 AND item_code = 1001;`
	Operação B começa.
	`SELECT * FROM inventory;`
	Operação B vê os dados atualizados pela operação A mesmo que não tenha encerrado.
	A leitura não repetida acontece quando uma consulta retorna dados que seriam diferentes se a consulta é repetida dentro da mesma transação. Isso pode ocorrer quando outras operações estão modificando os dados de que uma operação está lendo. Exemplo:
	Operação A começa.
	`SELECT * FROM inventory WHERe store_no = 10 AND item_code = 1001;`
	Operação B começa.
	`UPDATE inventory SET qtd = qtd – 50 WHERE store_no = 10 AND item_code = 1001;`
	Transação B visualiza atualizações pela transação A. Quando operação A emite a mesma declaração, irá produzir um resultado diferente.

Leitura fantasma	Uma leitura fantasma de registros ocorre quando aparece um conjunto a ser lido por outra operação. Isso pode ocorrer quando outras transações inserem linhas que satisfaçam a cláusula WHERE de uma outra operação da declaração. Exemplo: Operação A começa. `SELECT * FROM inventory WHERe qtd < 1000;` Operação B começa. INSERT INTO inventory VALUES(10, 1004, 1500, 100); Operação B insere uma linha que daria resposta a consulta na operação A. No entanto, só será incluído nos resultados de uma operação quando a consulta é re-executada.

Tabela 5: Anomalias na operação

A operação de níveis isolada é uma maneira de especificar se nestas operações são permitidas anomalias ou não. Isto afeta a quantidade de dados bloqueados por uma operação específica. O Quadro 4 mostra o sentido do isolamento de níveis e que anomalias são possíveis sob o nível de isolamento.

Nível de isolamento para JDBC (do objeto Connection)	Descrição
TRANSACTION_SERIALIZABLE	Isto significa que as transações são tratadas como se ocorressem em série (um após o outro) em vez de concorrentes. Bloqueios são emitidos à prevenção de todas as anomalias de operação.
TRANSACTION_REPEATABLE_READ	Bloqueios são emitidos para evitar má leitura e não leitura repetitiva, mas não lê fantasmas. Isto não bloqueia a emissão gama escolhe.
TRANSACTION_READ_COMMITTED	Bloqueios são emitidos para evitar má leitura. Este é o procedimento padrão do isolamento de nível.
TRANSACTION_READ_UNCOMMITTED	Para SELECT INTO, FETCH com cursor somente de leitura, seleção utilizada em INSERT, selecção completo / subquery em UPDATE / DELETE, ou selecionar pleno , permite que: ● qualquer linha que é lida durante a unidade de trabalho a ser alterada por outros processos de aplicações. ● qualquer linha que foi alterada por outro processo de aplicação a ser lido mesmo se a mudança não tenha sido feito por processo de aplicações. Para outras operações, as regras aplicáveis ao READ COMMITTED aplica-se igualmente a UNCOMMITTED.

Tabela 6: Descrições de Isolamento de Nível

No mais alto nível de isolamento, as alterações no banco de dados se tornam visíveis apenas quando a operação for encerrada. Quanto maior o nível de isolação, mais cuidado é necessário para evitar conflitos. Evitando conflitos por vezes significa bloquear as transações. Menor nível de isolação permite uma maior concorrência.

Para definir níveis de isolamento, você pode utilizar qualquer um dos seguintes procedimentos:

1. Use o método `setTransactionIsolation()` do objeto `Connection`. A seguir estabeleça o nível de isolamento do objeto `theConnection` para TRANSACTION_SERIALIZABLE.

   ```
   theConnection.setTransactionIsolation(theConnection.TRANSACTION_SERIALIZABLE);
   ```

2. Use o comando SET ISOLATION. Depois use SET ISOLATION do JavaDB.

   ```
   SET ISOLATION serializable;
   ```

   A sintaxe para comandos SET ISOLATION é mostrada a seguir:

```
SET [ CURRENT ] ISOLATION [ = ] {
 UR | DIRTY READ | READ UNCOMMITTED
 CS | READ COMMITTED | CURSOR STABILITY
 RS |
 RR | REPEATABLE READ | SERIALIZABLE
 RESET
}
```

5.2. Locks and Lock Granularity

Existem três tipos de bloqueios que JavaDB suporta. São eles:

1. **Bloqueio exclusivo**. A operação é dada em um exclusivo bloqueio se a transação modifica os dados do item. Isto permanece no local até que a transação emite bloqueio ou reversão.
2. **Bloqueio dividido**. A transação dada é uma bloqueio dividido se a transação lê o item sem modificar os dados de forma que quando uma outra operação tenta ler os mesmos dados, será autorizada a fazê-lo. No entanto, se uma outra operação tenta alterar os dados, ele terá que esperar até que o bloqueio seja liberado. A duração do bloqueio compartilhado dependerá do nível de isolação definido.

- Para TRANSACTION_READ_COMMITTED, o bloqueio compartilhado é liberado quando se move para a próxima linha.

- Para TRANSACTION_SERIALIZABLE ou TRANSACTION_REPEATABLE_READ, o bloqueio partilhado é liberado quando a transação encerra ou retorna.
- Para TRANSACTION_READ_UNCOMMITTED, não solicitará qualquer bloqueio.

1. **Bloqueios de atualização**. Este tipo de bloqueio é dado quando um cursor usuário-definido atualizado (criados utilizando a cláusula FOR UPDATE ou usando concurrency modo ResultSet.CONCUR_UPDATABLE) lê os dados. É convertida para um exclusivo bloqueio quando um cursor usuário definir-atualizar atualiza os dados. Se o cursor não atualizar os dados, quando a próxima linha é acessada pela operação de uma TRANSACTION_READ_COMMITTED ou TRANSACTION_READ_UNCOMMITTED, o bloqueio é liberado. Cursores serão debatidos na próxima seção.

A tabela a seguir mostra as compatibilidades de bloqueio (Tabela 4).

	Compartilhado	Atualizações	Exclusivo
Compartilhado	✓	✓	✗
Atualizações	✓	✗	✗
Exclusivo	✗	✗	✗

Tabela 7: Compatibilidade de bloqueios

Bloqueio de alcance

A quantidade de dados que pode ser bloqueada é conhecido como bloqueio alcance. Ela pode variar. Há três âmbitos de eclusas.

1. Bloqueio a nível de registro. Um método pode bloquear somente um registro por vez. Para o bloqueio à nível de registro, as regras são as seguintes:
   - Para TRANSACTION_REPEATABLE_READ, os bloqueios são desativados no momento em que a transação é finalizada.
   - Para TRANSACTION_READ_COMMITED, o bloqueio é ativado quando a transação acessa o registro, i.e., o registro corrente é bloqueado. O bloqueio é desativado quando a transação passa para o próximo registro.
   - Para TRANSACTION_SERIALIZABLE, o bloqueio é ativado para todo o conjunto de registros antes mesmo da transação acessar este conjunto. O bloqueio é desativado depois que todos os registros forem processadas.
   - Para TRANSACTION_READ_UNCOMMITED, nenhum bloqueio de registro é requisitado.
2. *Bloqueio de Intervalo.* Um método pode bloquear um intervalo de registros (intervalo de bloqueio). Para este tipo de bloqueio, as regras são as seguintes:
   - Para qualquer nível de isolamento, o sistema bloqueia todos os registros no resultado, mais um intervalo de registros para atualizações (updates) ou exclusões (deletes).
   - Para TRANSACTION_SERIALIZABLE, o sistema bloqueia todos os registros no resultado, mais todo um intervalo de registros na tabela para um SELECT, prevenindo leituras não-repetidas e fantasmas.

Por exemplo, se um comando SELECT especifica os registros na tabela de inventário onde a quantidade está ENTRE dois valores, digamos que 1000 e 2000, o sistema bloqueia todo o intervalo de registros entre estes dois valores para prevenir que outra transação insira, exclua ou atualize os dados neste intervalo.

Observe que um índice deve existir para bloqueios de intervalo. Caso o índice não exista, o sistema bloqueia toda a tabela.

3. *Bloqueio de Tabela.* Toda a tabela é bloqueada.

Bloqueio e Performance

Bloqueios à nível de registro melhoram a concomitância de sistemas multi-usuário. Entretanto, um grande número de bloqueios de registro podem comprometer a performance. Em alguns

SGDBs, existem componentes otimizadores para melhorar a performance do sistema no uso de bloqueios. Nesta sessão, discutiremos como o JavaDB Optimizer toma algumas decisões sobre a escalada de bloqueio a nível de registros até bloqueio a nível de tabelas por razões de desempenho.

Bloqueios baseados em transações

O objetivo das decisões do JavaDB é a concomitância. Sempre que possível, é escolhido o bloqueio a nível de registro. Entretanto, este tipo de bloqueio usa muitos recursos e pode ter um impacto negativo no desempenho. Às vezes o bloqueio a nível de registro não provê muito mais concomitância que bloqueios a nível de tabela. Nessas situações, o sistema pode escalar o esquema de bloqueio a nível de registro para bloqueio a nível de tabela para melhorar o desempenho.

Durante uma transação, o JavaDB rastreia o número de bloqueios para todas as tabelas, e quando este número excede um valor mínimo (que você pode configurar), é feito uma escalada para pelo menos uma das tabelas envolvidas, de bloqueio a nível de registros para bloqueio a nível de tabelas. Para realizar essa escalada em cada tabela que tenha um grande número de bloqueios, o JavaDB tentará obter o bloqueio mais relevante. Se o sistema puder bloquear a tabela imediatamente, toda a tabela é bloqueada e todos os bloqueios de registro são liberados. Caso contrário, os bloqueios de registro permanecem intactos.

A decisão baseada na transação em tempo de execução é independente de qualquer decisão de compilação. Se quando o valor mínimo da escalada for excedido e o JavaDB não obteve qualquer bloqueio de tabela porque seria necessário esperar, a próxima tentativa de escalada de bloqueio é atrasada até que o número de bloqueios pendentes tenha aumentado consideravelmente, por exemplo, de 5000 para 6000.

O JavaDB possui a propriedade *derby.locks.escalationThreshold* que determina o valor mínimo para um número de registros atingidos por uma tabela em particular, acima do que o sistema vai utilizar para escalar até bloqueios a nível de tabela. O valor padrão para esta propriedade é de 5000.

Bloqueando uma tabela pela duração de uma transação

Tabelas podem ser bloqueadas explicitamente utilizando o método LOCK TABLE. Isso é útil se você sabe por antecedência que uma tabela inteira deve ser bloqueada e quer poupar os recursos necessários para obter bloqueios de registro até que o sistema realize a escalada de bloqueios. A sintaxe do método LOCK TABLE é:

```
LOCK TABLE table-Name IN { SHARE | EXCLUSIVE } MODE
```

Este método lhe permite adquirir explicitamente um bloqueio de tabela compartilhado ou exclusivo na tabela especificada. O bloqueio da tabela dura até o fim da transação corrente. Depois que a tabela é bloqueada, seja qual for o modo, uma transação não pode adquirir nenhum bloqueio a nível de registro subseqüente numa tabela. Por exemplo, se uma transação bloqueia toda a tabela inventory em modo compartilhado para ler seus dados, e se um método em particular precisar bloquear um registro em modo exclusivo para que o mesmo seja atualizado, o bloqueio a nível de tabela em inventory força um bloqueio exclusivo para toda a tabela.

As Regras do JavaDB para Escalada de Bloqueio

Se um bloqueio for necessário, seja a nível de registros ou a nível de tabelas, e este bloqueio não for requisitado pelo usuário, o JavaDB toma uma decisão baseada nas seguintes regras.

- Para comandos SELECT sendo executados em TRANSACTION_READ_COMMITED, o JavaDB sempre escolhe o bloqueio a nível de registros.
- Quando o método faz uma busca em toda a tabela ou índice e o critério acima não é encontrado, o sistema escolhe o bloqueio a nível de tabela. (O sistema faz uma busca em toda a tabela sempre que escolhe uma tabela como o caminho de acesso).
- Quando um método faz uma busca parcial no índice, o sistema utiliza bloqueio a nível de registro até que um número de registros atingidos numa tabela cheguem ao valor mínimo de bloqueio para escalada.

- Para comandos SELECT, UPDATE e DELETE, o número de registros afetados é diferente do número de registros lidos. Se o mesmo registro é lido mais de uma vez, ele é considerado como afetado somente uma vez. Cada registro dentro de uma tabela de uma ligação pode ser lido muitas vezes, mas só pode ser afetado uma vez.

5.3. ResultSets e Cursores

Inserts, updates e deletes sempre se comportam da mesma forma, não importando qual o nível de isolamento. Somente o comportamento de um SELECT pode variar. Nesta sessão iremos nos concentrar em cursores atualizáveis.

Um resultado (result set) mantém um cursor, que aponta para seu registro atual de dados. Este pode ser utilizado para execuções em etapas e processar registros um por um. Como revisão, considere o código mostrado no Texto 5, que realiza um simples SELECT e então processa cada linha de dados da tabela.

```
private void populateItemData(){
  List<Item> theItemList = null;
  String selectStmt = "SELECT item.code, item.description, qtd, op_level " +
    "FROM item, inventory " +
    "WHERE inventory.item_code = item.code " +
    "AND inventory.store_no = " + this.storeNumber;
  try{
    this.theStatement = this.theConnection.createStatement();
    this.theResultSet = this.theStatement.executeQuery(selectStmt);
    if (this.theResultSet != null){
      theItemList = new ArrayList<Item>();
      while(this.theResultSet.next()){
        Item theItem = new Item();
        theItem.setCode(this.theResultSet.getInt(1));
        theItem.setName(this.theResultSet.getString(2));
        theItem.setqtd(this.theResultSet.getInt(3));
        theItem.setLevel(this.theResultSet.getInt(4));
        theItemList.add(theItem);
      }
    }
    this.theStore.setItemList(theItemList);
    this.theResultSet.close();
    this.theStatement.close();
  } catch (SQLException sqlx){
    sqlx.printStackTrace();
  }
}
```

Observe que o *auto-commit* está configurado para ON, já que este é o padrão do JDBC. Um objeto ResultSet (`theResultSet`) padrão é utilizado. Este resultado mantém o cursor que aponta para o registro atual de dados. O cursor se move uma linha para baixo toda vez que o método `next()` é invocado.

```
theResultSet.next();
```

O movimento do cursor, também chamado de rolagem, é de mão única. Ele se move para frente. Este é o padrão. Quando o *auto-commit* é um, depois do último registro, o método é considerado como completo e transação é aplicada (*commit)*. Observe também que, uma vez que a transação é aplicada, os bloqueios compartilhados são liberados.

O tipo de cursor que é mostrado no código de exemplo é chamado de *cursor somente leitura* ***(read-only cursor)***. É um cursor que aponta para registros que podem ser lidos mas não atualizados. Um simples SELECT teria cursores somente leitura.

Entretanto, um pode definir um ***cursor atualizável***. É um cursor que aponta para registros que podem ser atualizados. Para definir um cursor atualizável, nós utilizamos a cláusula FOR-UPDATE com nosso comando SELECT, e utilizamos os métodos do objeto `ResultSet`, como `updateRow()`, `deleteRow()` e `insertRow()`.

A cláusula FOR UPDATE especifica que o cursor deve ser atualizável e obriga uma checagem durante a compilação na qual o próximo SELECT cumpre os requerimentos para um cursor atualizável. A sintaxe da cláusula FOR UPTADE é:

```
FOR {
  READ ONLY | FETCH ONLY |
  UPDATE [ OF Simple-column-Name [ , Simple-column-Name]* ]
}
```

Simple-column-name se refere aos nomes visíveis para a tabela especificada na cláusula FROM na *query* subjacente. A utilização da cláusula FOR UPDATE não é obrigatória para se obter um ResultSet atualizável através do JDBC. Enquanto o método utilizado para gerar um ResultSet pelo JDBC cumprir o que é requerido para um cursor atualizável, é o suficiente para o Statement do JDBC, que gera o ResultSet do próprio, obter o modo concomitante ResultSet.CONCUR_UPTABLE para que o ResultSet seja atualizável. O otimizador é capaz de utilizar um índice mesmo que a coluna no índice esteja sendo atualizada. Para fazer com que um cursor seja atualizável usando o JDBC, precisamos especifica-lo durante a criação do método. Um exemplo é mostrado abaixo:

```
theStatement = theConnection.createStatement
  (ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
```

Neste exemplo, `theStatement` teria um cursor que se move apenas para frente e que seria atualizável.

Simples assim, cursores SELECT de apenas uma tabela podem ser atualizáveis. O método SELECT para ResultSets atualizáveis tem a mesma sintaxe que o método SELECT para cursores atualizáveis.

Para gerar cursores atualizáveis:

- O SELECT não pode ter uma cláusula ORDER BY.
- O SELECT na query subseqüente não pode ter:
  - DISTINCT
  - Aggregates
  - Cláusula GROUP BY
  - Cláusula HAVING
  - Cláusula ORDER BY
- A cláusula FROM na *Query* subseqüente não pode ter:
  - mais de uma tabela em sua cláusula FROM
  - nada mais que um nome de tabela
  - subqueries

Tipos de Cursores Atualizáveis

1. *Cursor atualizável de mão única.* Este é um tipo de cursor que só se move numa direção – para frente – ao mesmo tempo em que atualiza os registros. Para criar este tipo de cursor no JDBC, o objeto `Statement` deve ser criado com o modo de concomitância ResultSet.CONCUR_UPDATE e com o tipo ResultSet.TYPE_FORWARD_ONLY. Um código de exemplo é mostrado no Texto 6.

```
String selectStmt = "SELECT item_code, qtd FROM inventory";
  selectStmt = selectStmt + " WHERE store_no = 20";
theStatement = theConnection.createStatement(
  ResultSet.TYPE_FORWARD_ONLY,
  ResultSet.CONCUR_UPDATABLE);
theResultSet = theStatement.executeQuery(selectStmt);
int newqtd = 0;
while (theResultSet.next()){
  newqtd = theResultSet.getInt("qtd") + 100;
```

```
  theResultSet.updateInt("qtd", newqtd);
  theResultSet.updateRow();
}
```

O comando SELECT, como é representado pela string `selectStmt` é um SELECT simples que faz com que o banco de dados para registros de inventário armazene o número 20. Entretanto, nós estamos criando `theStmt` como um cursor atualizável utilizando o ResultSet.CONCUR_UPDATE. Esta é a razão pela qual podemos utilizar os métodos `updateInt()` e `updateRow()` do objeto `ResultSet`. O método `updateInt()` associa o novo valor da quantidade com a coluna `qtd` da tabela `inventory` para o registro atual. Para fazer alterações, o método `updateRow()` é invocado.

Nós utilizamos o método `updateInt()` desde que a coluna `qtd` seja de tipo inteiro. Se a coluna for de tipos, digamos, VARCHAR ou CHAR, nós utilizaríamos o método `updateString()`. A utilização do método apropriado `updateXXX()` depende do tipo de dado da coluna na qual o método vai agir.

Observe que nós definimos o cursor como FORWARD ONLY – o cursor se move numa única direção. O cursor se move desde o primeiro registro do `theResultSet` até seu último. Por esta razão, nós podemos utilizar o método `next()`, que faz com que o cursor aponte para o próximo registro do `theResultSet`.

O exemplo a seguir exclui os registros:

```
Statement theStatement = null;
ResultSet theResultSet = null
String selectStmt = "SELECT item_code, qtd FROM inventory";
theStatement = theConnection.createStatement(
  ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
theResultSet = theStatement.executeQuery(selectStmt);
while (theResultSet.next()){
  if (theResultSet.getInt("STORE")==30){
    theResultSet.deleteRow();
  }
}
```

Novamente, o `selectStmt` é um simples SELECT que retorna do banco de dados registros referentes ao inventário. `theStmt` é criado utilizando um cursor FORWARD ONLY atualizável. Utilizamos o método `next()` para mover o cursor até a próxima coluna. O cursor se move através de todo o resultado, verificando se o número de *store* é 30. Quando essa condição é verdadeira, deletamos o registro ao chamar o método `deleteRow()` do `theResultSet`.

Depois que um *update* ou *delete* é executado em um FORWARD ONLY, o cursor não está mais no registro recém atualizado ou excluído, mas sim imediatamente antes do próximo registro do resultado. Isso significa que é necessário mover o cursor até o próximo registro antes de executar qualquer outra operação. Com isso, as alterações feitas através de um *update* ou *delete* nunca serão visíveis. Entretanto, se o registro for inserido, ele pode ser visível.

O código a seguir mostra como inserir um registro utilizando um cursor atualizável:

```
Statement theStatement = null;
ResultSet theResultSet = null;
String selectStmt = "SELECT name, address FROM store";
theStatement = theConnection.createStatement(
       ResultSet.TYPE_FORWARE_ONLY, ResultSet.CONCUR_UPDATABLE);
theResultSet = theStatement.executeQuery(selectStmt);
theResultSet.moveToInsertRow();
theResultSet.updateString("NAME", "GangStore in Philadelpia");
theResultSet.updateString("ADDRESS","Philadelpia" );
theResultSet.insertRow();
theResultSet.moveToCurrentRow();
```

Neste exemplo, movemos o cursor para inserir um registro no resultado, que é especificado pelo método:

```
theResultSet.moveToInsertRow();
```

Então passamos os valores do registro que será inserido. Neste caso, utilizamos o método `updateXXX()` apropriado, por exemplo, `updateString()` em NAME e ADDRESS porque estes são do tipo VARCHAR. Um registro é inserido especificando o método:

```
theResultSet.insertRow();
```

Antes que qualquer outra operação com registros possa ser feita, é necessário que o cursor seja movido para o registro atual, como é mostrado no método:

```
theResultSet.moveToCurrentRow();
```

Quanto um registro é inserido, toda coluna que não aceita NULL e não possui um valor padrão deve ter um valor passado na hora do *insert*. Se o registro inserido satisfizer o predicado da *query*, ela pode se tornar visível no resultado.

2. *Cursor Atualizável de Duplo Sentido.* O cursor do resultado pode tanto atualizar os registros quando se mover. Para criar um cursor móvel e atualizável, o objeto `Statement` deve ser criado utilizando o modo de concomitância ResultSet.CONCUR_UPDATABLE e modo ResultSet.TYPE_SCROLL_INSENSITIVE. Como ilustração, considere o código a seguir:

```
String selectStmt = "SELECT name, address FROM store";
theStatement = theConnection.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
theResultSet = theStatement.executeQuery(selectStmt);
theResultSet.absolute(2);
theResultSet.updateString("ADDRESS", "Livingstone, West Virginia");
theResultSet.udpateRow();
```

3. No exemplo, estamos recebendo registros da tabela *store*. `theStatement` é criado utilizando um cursor móvel e atualizável. Quando um cursor como este é utilizado num resultado, podemos, na verdade, utilizar o método `absolute()` para mover o cursor até o enésimo registro. Neste caso, queremos mover o cursor até o segundo registro, como é especificado pelo método:

```
theResultSet.absolute(2);
```

4. Agora podemos atualizar o endereço utilizando o método `updateString()` desde que a coluna ADDRESS seja do tipo VARCHAR.

   Outro exemplo é mostrado no código a seguir:

```
String selectStmt = "SELECT name, address FROM store";
theStatement = theConnection.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
theResultSet = theStatement.executeQuery(selectStmt);
theResultSet.last();
theResultSet.relative(-2);
theResultSet.deleteRow();
```

5. Neste exemplo, utilizamos o método `relative()` para mover o cursor. A diferença entre os métodos `absolute()` e `relative()` é que o último se move baseado na posição atual do cursor. De forma diferente como `absolute()`, o cursor se move baseado na posição do registro no resultado. No exemplo, o cursor é movido primeiro ao ponto do último registro no resultado, o que é especificado pelo método:

   ```
   theResultSet.last();
   ```

   Depois o cursor é movido para o segundo do último registro, o que é especificado pelo método:

   ```
   theResultSet.relative(-2);
   ```

Então o registro naquela posição é excluído.

5.4. Deadlocks

Um bloqueio é uma situação em que duas ou mais transações estão esperando para que uma outra libere bloqueios. Nessa sessão, discutiremos meios de como evitar estes bloqueios.

Evitando *deadlock*

Aqui vão algumas dicas de como evitar bloqueios com transações que estão definidas.

1. Para diminuir os riscos de bloqueios, utilize tanto bloqueios a nível de registro como nível de isolamento TRANSACTION_READ_COMMITED. Essas são as configurações padrão do JavaDB.
2. Utilize lógica de aplicação consistente. Exemplo, transações que acessam as tabelas *orders* e *inventory* devem sempre acessar essas tabelas na mesma ordem. Dessa forma, a segunda transação simplesmente espera até que a primeira libere o bloqueio em *orders* antes de iniciar seu processamento. Quando a primeira transação libera o bloqueio, a segunda pode começar.
3. Utilize o método LOCK TABLE. Uma transação pode tentar bloquear uma tabela de modo exclusivo e assim prevenir que outras transações obtenham um bloqueio compartilhado numa tabela.

Detecção de *deadlock*

O JavaDB detecta quando transações estão envolvidas num bloqueio quando descobre que esta transação está esperando mais tempo que o especificado para obter um bloqueio, o que é conhecido como tempo esgotado até o bloqueio.

O JavaDB analisa a situação para bloqueios. Ele tenta determinar quantas transações estão envolvidas no bloqueio. Normalmente, abortar uma transação quebra o bloqueio, e o JavaDB pega a transação que tiver menos bloqueios como sendo a vítima porque pressupõe que esta foi a transação que menos realizou tarefas (Entretanto, este pode não ser o caso porque é possível que uma transação tenha feito uma escalada de um bloqueio a nível de registro até um bloqueio a nível de tabela, e mesmo tendo um número pequeno de bloqueios pode ter realizado várias tarefas no banco de dados). Quando o JavaDB aborta a transação, ele chama uma exceção SQLException com o SQLState de 40001. Ele dá identificadores à transação, os comandos e o status do bloqueio envolvido num bloqueio.

Uma aplicação pode ser programada para lidar com bloqueios. No código de exemplo testamos a SQLException com SQLStates igual a 40001 (por exemplo, tempo esgotado até o bloqueio). No caso de um bloqueio, seria melhor reiniciar a transação.

Neste exemplo, a transação move o estoque (stock) de uma loja (store) para outra. Se essa transação for pega num bloqueio, o JavaDB chama uma SQLException. O bloqueio pego checa se o SQLState é 40001, o que significa bloqueio. Se for o caso, a transação é reiniciada. Antes que a SQLException seja invocada, quaisquer alterações feitas no banco de dados são canceladas. Deste modo, o banco de dados é colocado num estado consistente antes que a transação seja reiniciada.

```
try {
  s6.executeUpdate("UPDATE inventory SET qtd = qtd - 100 " +
    "WHERE store_no = 20 AND item_code = 1001");
  s6.executeUpdate("UPDATE inventory SET qtd = qtd + 100 " +
    "WHERE store_no = 10 AND item_code = 1001");
} catch (SQLException se) {
  if (se.getSQLState().equals("40001")) {
    System.out.println( "Will try the transaction again.");
    s6.executeUpdate("UPDATE inventory SET qtd = qtd - 100 " +
      "WHERE store_no = 20 AND item_code = 1001");
    s6.executeUpdate("UPDATE inventory SET qtd = qtd + 100 " +
      "WHERE store_no = 10 AND item_code = 1001");
  } else
    throw se;
```

```
}
```

5.5. Recuperação de bancos de dados

Quando um comando SQL gera uma exceção, esta resulta num cancelamento em tempo de execução. Um ***cancelamento em tempo de execução (runtime rollback)*** é um cancelamento gerado pelo sistema de um comando ou transação pelo JavaDB.

O JavaDB faz um cancelamento em tempo de execução nas seguinte exceções.

1. **Extremamente severa** – como espaço em disco insuficiente ou desligamento do sistema. Neste caso, o JavaDB faz um cancelamento na próxima vez em que o sistema for inicializado.
2. **Severa** – como bloqueio ou falha na transação, o JavaDB volta para o início da transação.
3. **Menos severa** – como erros de sintaxe, faz com que o JavaDB cancele apenas o comando.

6. Exercício

1. Utilizando o sistema distribuído e-Lagyan, crie uma Aplicação Java que simule uma transação de cartão telefônico, que defina os seguintes passos como uma unidade lógica de trabalho.
   - Insere um registro na tabela de transação telefônica que mostra o número do telefone do emissor, o número do telefone do receptor, a data da transação e o número do cartão do telefone.
   - Marcar que o cartão foi utilizado na tabela de cartões.

     Fazer o balanço da conta do emissor.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Instituto Gaudium
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.