

Módulo 9

Banco de Dados



Lição 1

Introdução a Sistemas de Bancos de Dados

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

Nesta lição veremos uma introdução a sistemas de bancos de dados. A primeira parte descreve o ambiente de banco de dados utilizando o conceito de Gerenciamento de Recursos de Informações ou IRM (*Information Resource Management*) em que as empresas tem necessidade de gerenciar seus dados e informações.

A segunda seção discute o processo de desenvolvimento de bases de dados utilizando o framework a Arquitetura de Sistema de Informação ou ISA (*Information System Architecture*). Isto ajudará a entender o lugar do banco de dados no processo de desenvolvimento e manutenção de sistemas de informação em uma empresa.

Ao final desta lição, o estudante será capaz de:

- Descrever o ambiente de banco de dados utilizando o conceito de Gerenciamento de Recursos de Informação
- Descrever a Arquitetura de Sistemas de Informação como um *framework* para construção de sistemas de informação
- Entender a importância do banco de dados dentro do desenvolvimento geral de sistemas de informação
- Discutir os processos de análise, design e implementação de um banco de dados

2. Ambiente de Bases de Dados

Com o passar dos anos, as empresas perceberam a importância do dado e da informação em suas operações diárias. IRM (Gerenciamento de Recursos de Informação) é o conceito de que a informação é um recurso corporativo muito importante e deve ser gerenciado utilizando alguns princípios básicos que são utilizados para gerenciar outros ativos da companhia como pessoal, equipamento e recursos financeiros.

A seguir vemos os princípios básicos do qual IRM é derivado:

1. Empresas utilizam recursos que fluem em seu ambiente
2. Meio ambiente. Provê recursos retornando ele para seu ambiente de origem
3. Existem dois tipos básicos de recursos para serem gerenciados, especificamente:
 - Recursos Físicos que são, por exemplo: pessoal, materiais, máquinas, entre outros
 - Recursos Conceituais como dados e informações
4. Com o crescimento das operações organizacionais, torna-se difícil gerenciar os recursos físicos utilizando observações. Portanto, gerentes de negócios são forçados a depender de recursos conceituais
5. Os mesmos princípios básicos utilizados para o gerenciamento de recursos físicos podem ser utilizados para gerenciar os recursos conceituais
6. O gerenciamento de dados e informações, envolve:
 - Aquisição de dados e informações antes que sejam necessárias
 - Medidas de segurança para proteger recursos contra invasão, uso indevido e destruição
 - Garantia de qualidade
 - Procedimentos de liberação de recursos quando não são mais necessários à organização
7. Compromisso organizacional é necessário para gerenciar os dados e informações.

Para implementar IRM, as seguintes funcionalidades são consideradas:

1. Gerenciamento de operações, tais como: agendamento, planejamento de capacidade, segurança de operações e desastres de recuperação de dados e informações
2. Garantia de qualidade para garantir que informações sejam fornecidas quando necessário
3. Gerenciamento de comunicações de LAN ou WAN
4. Gerenciamento de recursos de dados tais como análise de dados, design de bancos de dados, administração de dados e administração do banco de dados
5. Gerenciamento de projeto
6. Planejamento de Sistemas de Informação corporativos
7. Desenvolvimento e manutenção de sistemas

Dados são fatos acerca de pessoas, objetos e eventos. Informação, por outro lado, é o dado que foi devidamente processado e apresentado num formulário para a interpretação humana, freqüentemente com o propósito de revelar tendências e padrões. Existem 5 tarefas envolvidas na conversão de dados em informações. São eles:

1. Aquisição
2. Armazenamento
3. Manipulação
4. Recuperação

5. Distribuição

Para dar suporte aos 5 itens citados acima é necessária a utilização de uma base de dados. Um banco de dados é uma coleção compartilhada de dados lógicos relacionados, de tal forma a prover informação para múltiplos usuários em uma organização. Existem duas arquiteturas genéricas de bancos de dados: bancos de dados centralizados e bancos de dados distribuídos.

2.1. Banco de Dados Centralizado

Em um banco de dados centralizado, os dados estão armazenados em um único local. Estes dados são acessados por meio de equipamentos de comunicação. Fornecem um mecanismo de controle de acesso e atualização dos dados melhor do que os bancos de dados distribuídos, porém são mais vulneráveis a falhas já que dependem da disponibilidade de recursos de um único local. Três exemplos de bancos de dados centralizados são discutidos logo abaixo:

1. **Banco de Dados de Computadores Pessoais.** Neste ambiente o banco é utilizado por um único usuário. Este usuário cria o banco de dados, atualiza e mantém os dados, produz os dados e gera relatórios ou informação com base nestes dados. Normalmente o banco de dados está alocado em um computador pessoal onde um ou um número limitado de aplicações acessam este banco. Este tipo de banco de dados pode ser encontrado em pequenos negócios. Um exemplo de aplicação pode ser o gerenciamento de um estoque.
2. **Bancos de Dados em Computadores Centrais.** Neste ambiente o banco de dados é normalmente localizado numa máquina central e é chamado de host. Os dados são acessados por terminais e meios de comunicações de dados. O computador é normalmente um mainframe ou servidor encontrado em grandes negócios que necessitam de um intenso acesso aos dados por um grande número de usuários. Típicas aplicações podem ser: sistemas de reservas aéreas, instituições financeiras e companhias de entrega.
3. **Bancos de Dados Cliente/Servidor.** Neste ambiente a arquitetura cliente-servidor é utilizada, onde muitos clientes podem compartilhar o serviço de um único servidor. O servidor é um software de aplicação que provê serviços (chamados de funções back-end, tais como impressões, gerenciamento de bancos de dados ou de arquivos, gerenciamento de comunicações, entre outros) para os clientes que os requisitam. Um cliente (que fornece as funções de *front-end*) é um software de aplicações que requisita o serviço de um ou mais servidores. O ponto forte de uma arquitetura cliente-servidor é permitir que a aplicação do cliente acesse dados gerenciados pelo servidor.

2.2. Bancos de Dados Distribuído

Um banco de dados distribuído é representado por um único banco de dados lógico que é separado fisicamente em vários computadores que podem estar situados em vários locais. Existem duas categorias genéricas.

1. **Bancos de dados homogêneos.** A tecnologia de bancos de dados utilizada é a mesma ou pelo menos compatível em todos os locais. Certas condições devem ser respeitadas antes de um banco de dados ser considerado homogêneo:
 - O sistema operacional utilizado em cada uma das localidades deve ser o mesmo ou pelo menos um que seja altamente compatível
 - O modelo de dados utilizado em cada uma das localidades deve ser o mesmo
 - O sistema gerenciador de banco de dados utilizado em cada uma das localidades deve ser o mesmo ou pelo menos que sejam altamente compatíveis entre si
 - Os dados em suas várias localidades devem possuir definições e formatos comuns
2. **Bancos de dados Heterogêneos.** A tecnologia utilizada varia e pode não ser a mesma em todas as localidades. Um banco de dados pode utilizar uma tecnologia de gerenciamento relacional enquanto que outro utilize arquivos convencionais ou antigos bancos de dados hierárquicos. Estes bancos de dados ficam conectados.

3. Arquitetura de Sistemas de Informação (ISA)

Para desenvolver sistemas de informação, um *framework* é utilizado para fornecer uma base para o planejamento estratégico, desenvolvimento e utilização de sistemas de informação que dão suporte a visão geral dos objetivos da organização. A arquitetura de sistemas de informação é um exemplo de *framework*. Representa um modelo conceitual ou plano que ilustra a estrutura dos sistemas de informação que são necessários para a organização. Fornece a base para planejamentos estratégicos e comunicações para a direção de toda a tecnologia da informação e o contexto principal para a tomada de decisão em uma área. A tabela 1 descreve este *framework*.

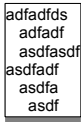
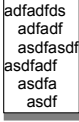
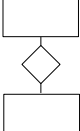

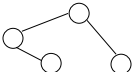
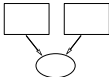
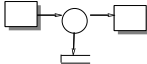
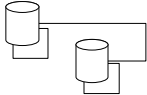


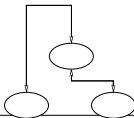
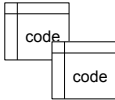
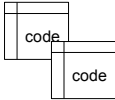
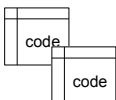
		Dado	Processos	Rede
1	Escopo do negócio	Lista as entidades importantes de um negócio 	Lista as funções que o negócio deve realizar 	Lista de localidades em que o negócio opera
2	Modelo do negócio	Entidade de negócio e seus inter-relacionamentos 	Decomposição de funções e processos 	Links de comunicação entre as localidades do negócio 
3	Modelo do sistema de informação	Modelo de negócios e seus inter-relacionamentos 	Fluxos entre os processos 	Distribuição em redes 
4	Modelo de tecnologia	Design do banco de dados 	Especificação dos processos 	Design de configuração 
5	Definição da tecnologia	Definição da estrutura e subestrutura do banco de dados 	Códigos do programa e blocos de código 	Definição de configuração 
6	Sistema de informação	Dados e informações	Aplicações	Configuração do sistema

Tabela 1: Arquitetura de Sistemas de Informação

Os três componentes principais neste *framework* são dados, processos e rede. Estes são as três colunas da tabela.

1. **Dados** consistem de entidades de dados e o relacionamento entre si. Representam o “o quê?” em um sistema de informação. Este é o componente com o qual bancos de dados são construídos.
2. **Processos** são seqüências de passos que convertem entradas em saídas (ou dados em informação). Representam o “como” em um sistema de informação.

3. **Rede** descreve a localização onde os dados são armazenados, onde os processos são realizados bem como a conexão entre as localidades. Representam o “onde” em um sistema de informação.

Cada linha representa uma camada arquitetural na construção de sistemas de informação de uma organização. Fornecem as 6 regras e perspectivas de cada camada.

1. **Escopo do Negócio** – provê uma visão geral da estratégia dos sistemas de informação de uma organização. Define o escopo, missão e direção do negócio e quais sistemas de informação lhes dá suporte. Uma lista de entidades importantes, funções necessárias para que o sistema possa realizar aquilo que deve ser feito e identificar onde o negócio da empresa é realizado. Os proprietários do negócio são responsáveis por definir o escopo, missão e direção do negócio
2. **Modelo de Negócio** – desenvolve os modelos que representam o escopo do negócio, missões e direções que o negócio irá tomar. Nesta camada, as entidades do negócio e seus inter-relacionamentos são definidos. A decomposição de funções e processos são identificadas. É definida as ligações entre as localidades de negócios. O arquiteto do sistema de informação é a pessoa que desenvolve estes modelos
3. **Modelo do Sistema de Informação** – desenvolve o modelo da informação que dá suporte ao negócio da organização. Nesta camada, os dados e seus relacionamentos são modelados em detalhes. Os fluxos entre as aplicações também são processados e definidos. A distribuição pela rede também é identificada. O designer é responsável pelo desenvolvimento do modelo da informação
4. **Modelo de Tecnologia** – converte o modelo do sistema de informação em um design que possa se adaptar às características e especificações da tecnologia. O projeto de um banco de dados, especificação de processos e configurações de rede são criados. O construtor desenvolve o design do sistema de informação
5. **Definição da Tecnologia** – converte os modelos de tecnologia em declarações para gerar o sistema de informação. O projeto do banco de dados é traduzido em esquema e sub-esquemas do banco de dados. As especificações de processo são codificadas como códigos de programa e blocos de controle. Os projetos de configuração da rede são traduzidos em definição de configuração. O contratante traduz modelos de tecnologia em códigos
6. **Sistema de Informação** – gerencia, utiliza e opera o sistema de informação como um todo. Neste ponto, os usuários do sistema utilizam os dados e informações através de aplicativos especificados na configuração de sistema

Para melhor utilizar este *framework*, duas regras simples são empregadas:

1. Cada processo é mapeado para o dado que o utiliza. Ambos, dados e processos, são mapeados para as localizações na rede ou objetos onde serão distribuídos. Isto ajuda na garantia de que os vários componentes serão integrados aos demais
2. A transformação de dados, processos e rede ocorrem simultaneamente de uma linha para a próxima. Esta regra evita inconsistência e retrabalho

Este curso irá se concentrar no COMPONENTE DE DADOS do *framework* da Arquitetura do Sistema de Informação (ISA) na construção de um banco de dados.

4. Metodologia de Engenharia da Informação

O *framework* ISA fornece um contexto de desenvolvimento e integração do sistema de informação. Sugere o tipo de processo no desenvolvimento de modelos em cada uma das camadas da arquitetura. Entretanto, não fornece uma forma para o desenvolvimento destes modelos. Portanto, uma organização deve utilizar uma ou mais metodologias e um conjunto de ferramentas de modelagem para desenvolver a representação arquitetural requisitada em cada perspectiva.

Uma metodologia define o “como” ou conjunto de passos para se realizar determinado objetivo,

juntamente com um conjunto de objetos de design que são manipulados para auxiliar o processo. Deve fornecer regras para ajudar a garantir que um conjunto consistente de padrões e procedimentos serão utilizados durante todo o processo de desenvolvimento e o sistema resultante se adequar aos objetivos especificados.

Uma variedade de ferramentas de modelagem, manuais ou automatizadas, são necessárias para auxiliar no desenvolvimento do sistema de informação.

1. **Engenharia de Software Auxiliada por Computador (CASE)** – São produtos de software que fornecem suporte automatizado para algumas partes do processo de desenvolvimento do sistema.
2. **CASE Integrado (I-CASE)** – Conjunto de ferramentas CASE que podem dar suporte a todas as fases do processo de desenvolvimento de um software.

Engenharia da Informação é uma metodologia formal que é utilizada na criação e manutenção de sistemas de informação. É um padrão *top-down* que se inicia com os modelos de negócio. A partir destes modelos, os modelos de dados e modelos de processos são derivados e então é criado o modelo de negócio. Porque utilizar Engenharia da Informação?

1. Desenvolvido com uma visão baseado na empresa que permite que a organização desenvolva um sistema de informação integrado
2. Baseado em dados ao invés de processos. Um modelo baseado em dados utiliza as seguintes etapas:
 - Identificação de entidades ou coisas que a organização deve gerenciar
 - Identificação de atributos, propriedades e características das entidades
 - Identificação dos relacionamentos entre as entidades
 - Identificação de regras de negócio que governam como as entidades são gerenciadas e utilizadas
 - Desenvolver o aplicativo ou programa baseado em como os dados estão sendo gerenciados e utilizados

Um modelo baseado em processos utiliza as seguintes etapas:

- Identificação e análise dos processos organizacionais
- Modelagem dos fluxos de dados entre os processos
- Especificação das entradas e saídas de dados
- Especificação da lógica necessária para converter dados de entrada em dados de saída
- Design dos arquivos de dados

Designers de Sistemas descobriram que um equilíbrio entre modelo baseado em dados e modelo baseado em processos são normalmente o mais apropriado.

3. É compatível com o *framework* ISA

A Metodologia da Engenharia de Informação é dividida em fases que podem ser mapeadas para o *framework* ISA. São normalmente chamadas de: fase de planejamento, fase de análise, fase de projeto e fase de implementação.

4.1. Fase de Planejamento

O objetivo desta fase é alinhar a tecnologia da informação às estratégias do negócio de uma organização. Para isto é necessário que haja uma cooperação entre os gerentes do negócio e os gerentes do sistema de informação. Existem três grandes passos na fase de planejamento. São eles:

1. Identificação de fatores de planejamento de estratégia que incluam os objetivos do negócio, fatores críticos de sucesso e problemas das áreas do negócio.
2. Identificação de objetos de planejamento corporativo. Estes objetos são:
 - Unidades da organização que consistem de vários departamentos ou outros

- componentes da organização
 - Localizações que mostrem os componentes organizacionais em mais de um único lugar
 - Funções de negócio relacionados a grupos de processos de negócio que dão suporte a alguns aspectos da missão da empresa e que não são os mesmos da unidade organizacional
 - Tipos de entidades
3. Desenvolver os modelos da empresa utilizando as técnicas e ferramentas de modelagem a seguir:
- Técnica de decomposição funcional que consiste na quebra das funções de uma organização em busca de níveis de detalhes cada vez maiores, utilizando o **Diagrama de Fluxo de Dados** para modelar as funções ou processos
 - Técnica de análise situacional que consiste no processo de análise e identificação de entidades que representam os dados importantes para a organização e o **Diagrama de Entidade e Relacionamento** utilizado para modelar a estrutura dos dados
 - Matriz de planejamento para vincular as funções identificadas na decomposição funcional com às entidades com o propósito de identificar órfãos. Por exemplo, identifica quais funções não fazem uso de nenhuma entidade ou entidades que não são utilizadas por nenhuma função

```
Lagyan Cards Incorporated
Business Goals:
1. Increase distribution of cards by 50% for the next three years.
2. Increase network of dealers nationwide by 10% for the next year.

Organizational Units:
1. Sales Department
2. Accounting Department
3. Financial Department
4. Business Centers

Business Functions:
1. Inventory of e-Prepaid Cards
2. Accounting of all financial transaction
3. Sales monitoring of e-Prepaid Cards

List of Entity Types
1. Customers, Direct Resellers, Dealers
2. e-Prepaid cards, phone cards and internet access cards
3. e-prepaid card transactions
```

Figura 1: Objetos de Planejamento Corporativo

A fase de planejamento gera vínculos com a camada de Escopo do Negócio do *framework* ISA.

4.2. Fase de Análise

Esta fase é também conhecida como a fase da Engenharia de Requisitos. O propósito é desenvolver especificações detalhadas para o sistema de informação requisitado pela organização. Essas especificações incluem suporte a decisões e sistemas de informações executivos bem como sistemas de processamentos transacionais. Cobre o estudo da atual situação do negócio bem como a determinação dos requisitos para o novo sistema. Lida com uma área de negócios por vez que é o agrupamento de funções e entidades coesas que dão base para o desenvolvimento do sistema de informação.

CrITÉRIOS para a definição da Área de Negócios

1. Área de negócios deve ser claramente delimitada.
2. Pequena o bastante para ser bem entendida e facilmente gerenciável.
3. Grande o suficiente para necessitar de bancos de dados compartilhados.

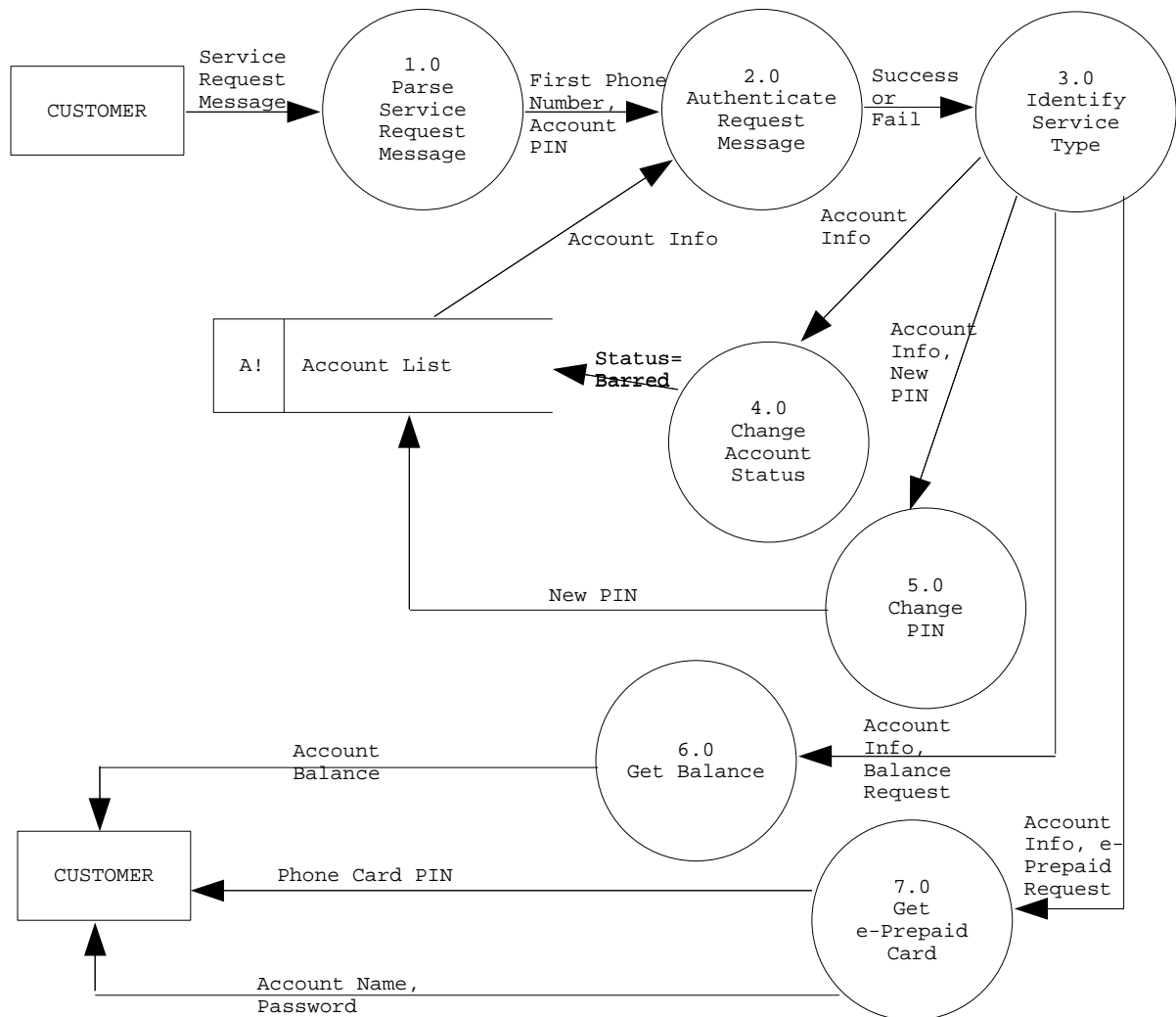


Figura 3: DFD Nível 0 – Serviço de Requisição do Sistema de Mensagem

A fase de análise está relacionada com as camadas **Modelo do Negócio** e **Modelo do Sistema de Informação** do *framework* ISA.

4.3. Fase de Projeto

O objetivo da Fase de Projeto é transformar o **Modelo Conceitual** e o **Modelo de Processo** desenvolvidos durante a Fase de Análise em modelos que se adequar a tecnologia a ser utilizada na implantação do sistema de informação. Dois modelos de projeto são criados, **Projeto de Banco de Dados** e **Projeto de Processo**.

Passos da Fase de Projeto

1. Desenvolvimento do Projeto do Banco de Dados

O propósito desse passo é mapear o **Modelo Conceitual de Dados** para o **Modelo de Implementação** que um Sistema Gerenciador de Banco de Dados (SGBD) pode processar com uma performance aceitável para todos os usuários da organização. Para construir o banco de dados dois projetos são criados.

- **Projeto Lógico do Banco de Dados** que mapeia o Modelo Conceitual desenvolvido na análise das estruturas específicas do SGBD. A figura 4 mostra um trecho de um projeto lógico de banco de dados.
- **Projeto Físico do Banco de Dados** é o mapeamento das estruturas do Projeto Lógico do Banco de Dados para estruturas de armazenamento físico, tais como, arquivos e entidades. Índices também são especificados assim como métodos de acesso e outros

fatores físicos. O objetivo principal de se ter o design físico do banco de dados é fornecer performance adequada para usuários de aplicações em termos de tempo de resposta, taxas de transferência, entre outros. Cópias de segurança e sua restauração são considerados no Projeto Físico.

ACCOUNT

CellPhoneNo	PIN	Balance	Limit	Status	Type
09192345678	1234	\$500,00	\$2.500,00	ACT	2
09174561234	2345	\$100,00	\$2.000,00	ACT	2
09205467234	4523	\$25.000,00	\$300.000,00	ACT	1
09165647342	7812	\$30.000,00	\$300.000,00	ACT	1

STATUS

CODE	DESCRIPTION
ACT	Active Account
BAR	Barred Account
TER	Terminated Account

TYPE

CODE	DESCRIPTION
1	Dealer Account
2	Direct Reseller

PHONECARDTRANSACTION

CELLPHONENO	CARDNO	LOADDATE	RECIPIENT
09205467234	2346253782	DEC-24-2006	9223456173
09205467234	8736237634	DEC-24-2006	9178746345

Figura 4: Exemplo de Projeto Lógico do Banco de Dados

2. Desenvolvimento do **Projeto de Processo**.

O propósito desse passo é especificar a lógica de cada um dos processos e incluir todas as referências para as entidades relevantes. Existem dois subpassos para o Projeto de Processo:

- Especificar a lógica detalhada de cada processo
- Desenhar interfaces de usuários que podem ser: menus, formulários e relatórios, entre outros

A figura 5 é um exemplo da especificação de um Projeto de Processo. O trecho da especificação com fonte vermelha deve ser tratado como uma transação.

Project Name: e-Lagyan Distribution System
 Company: Lagyan Cards Incorporated

Process Specifications: Phone Card Service Request

INPUT: PAccountNo, PBalance, VFirstCellNo, VSecondCellNo,
 VService, VDenomination

```

IF PBalance < VDenomination THEN
  MESSAGE "Insufficient Balance"
  Terminate Transaction
ELSE
  SELECT InventoryCount
  INTO PCount
  FROM CellCardSupply
  WHERE Code IN (SELECT Code
                  FROM CellCardType
                  WHERE Denomination = VDenomination
                  AND SPID = VService)

  IF (PCount < 0) THEN
    MESSAGE "No Card Available" TO VFirstCellNo
  ELSE
    SELECT FirstAvailableSerialNo, PIN, CardType
    INTO PSerialNo, PPIN, PCardType
    FROM CellCard
    WHERE CardType IN (SELECT Code
                       FROM CellCardType
                       WHERE SPID = VService
                       AND Denomination = VDenomination)
    AND Status = "VALID" or "UNSOLD"
    IF VSecondCellNo <> NULL THEN
      INSERT INTO CellCardTrans VALUES
        (PAccountNo, PSerialNo, SYSDate, VSecondCellNo)
    ELSE
      INSERT INTO CellCardTrans VALUES
        (PAccountNo, PSerialNo, SYSDate, VFirstCellNo)
    ENDIF
    UPDATE Account SET
      Balance = Balance - Denomination
    WHERE AccountNo = PAccountNo
    UPDATE CellCardSupply SET
      Count = Count - 1
    WHERE Code = PCardType
    UPDATE CellCard SET
      Status = 'Invalid' or 'Sold'
    WHERE SerialNo = PSerialNo
    IF VSecondCellNo <> NULL THEN
      SEND PPIN to VSecondCellNo
    ELSE
      SEND PPIN to VFirstCellNo
    ENDIF
  ENDIF
ENDIF

```

Figura 5: Especificação do Projeto de Processo dos cartões telefônicos e-Prepaid

4.4. Fase de Implementação

O propósito da fase de implementação é construir e instalar o sistema de informação de acordo com os planos e designs. Isto envolve uma série de passos que acarretará a operacionalização do sistema de informação que incluem criar definições de banco de dados, criar códigos de programas, testar o sistema, desenvolver procedimentos operacionais e documentação.

5. Componentes para um SGBD

Sistema Gerenciador de Banco de Dados (SGBD) são softwares altamente sofisticados e complexos que tem por objetivo fornecer serviços para gerenciar dados de uma organização. Nessa seção, os tipos genéricos de funções e serviços serão discutidos e uma possível arquitetura de um SGBD será apresentada.

5.1. Serviços e Funções de um SGBD

Codd¹ defini oito serviços que devem ser fornecidos por um SGBD de alta-escala.

1. Um SGBD deve fornecer aos usuários uma forma de armazenar, recuperar e atualizar dados no banco de dados. Essa é a função fundamental de um SGBD.
2. Um SGBD deve fornecer um catálogo com descrição de itens de dados armazenados e acessíveis aos usuários. A chave para esse serviço é o **catálogo de sistema** ou **dicionário de dados**, que é um repositório de informações descrevendo os dados no banco de dados. É referenciado como dados sobre dados ou **meta-dados**.
3. Um SGBD deve fornecer um mecanismo que assegure que todas as atualizações correspondentes a uma dada transação sejam feitas ou que nenhuma delas seja feita. A chave para esse serviço é o conceito de transação. Uma **transação** é uma série de ações que acessam ou modificam o conteúdo do banco de dados. Quando executada, deve garantir que o banco de dados esteja sempre num estado estável e consistente.
4. Um SGBD deve fornecer um mecanismo que assegure que o banco de dados seja atualizado corretamente quando múltiplos usuários atualizam o banco de dados concorrentemente. A chave para esse serviço é permitir que muitos usuários acessem o banco de dados simultaneamente. O acesso concorrente é relativamente fácil quando os usuários estão apenas lendo os dados. Entretanto, quando os usuários estão acessando o banco de dados simultaneamente e pelo menos um deles está escrevendo dados, podem ocorrer interferências que podem resultar em inconsistências. O SGBD deve garantir que acessos concorrentes não deixarão o banco de dados num estado inconsistente.
5. Um SGBD deve fornecer um mecanismo para recuperação do banco de dados no caso de dados serem danificados de alguma maneira. Isso está relacionado com as transações. Quando uma transação falhar, o SGBD deve retornar ao estado consistente que estava antes da transação ser executada.
6. Um SGBD deve fornecer um mecanismo para garantir que apenas usuários autorizados tenham acesso ao banco de dados. A chave para esse serviço é a segurança, que se refere à proteção do banco de dados contra acessos não autorizados, intencionais ou não.
7. Um SGBD deve ser capaz de se integrar com softwares de comunicação.
8. Um SGBD deve fornecer um meio de garantir que os dados e suas mudanças sigam certas regras. A integridade de dados se refere-se à exatidão e consistência dos dados armazenados no banco de dados. É usualmente expressa em termos de restrições (*constraints*) que são regras de consistência que o banco de dados não deve violar. Também é conhecido como implementação de regras de negócio.

Outros serviços a serem considerados e fornecidos por um SGBD, são:

1. Um SGBD deve incluir facilidades dar suporte à independência dos programas da estrutura do banco de dados. A chave para isso é o conceito de visões (*views*) e subesquemas.
2. Um SGBD fornece um conjunto de serviços utilitários:
 - facilidades de importação que carreguem o banco de dados utilizando arquivos, e facilidades de exportação que descarregam o banco para arquivos.
 - facilidades de monitoramento da utilização e operação do banco de dados

¹ Codd E. F. The 1981 ACM Turing Award Lecture: Relational Database: A Practical Foundation for Productivity. Comm. ACM, 25(2), 109-117

- análises de performance e estatísticas de utilização
- facilidades para reorganização de índices e suas sobrecargas
- *garbage collection* e realocação para remover fisicamente os registros eliminados para consolidar o espaço liberado e realocá-lo quando houver necessidade

5.2. Arquitetura de Banco de Dados

Não é possível generalizar a estrutura de componentes de um SGBD, pois isto varia de sistema para sistema. No entanto, é útil entender o sistema de banco de dados visualizando seus componentes e como eles se relacionam. Figura 6¹ é uma arquitetura possível de SGBD.

Um SGBD é modularizado em diversos componentes de softwares relacionados. Cada componente foi desenvolvido para executar operações específicas. O diagrama mostra como cada componente interage com os outros. A seguir listamos os componentes.

- **Processador de Consultas.** É considerado um dos principais componentes do SGBD. Transforma consultas em uma série de instruções de baixo-nível que são direcionadas ao software gerenciador do banco de dados
- **Gerenciador de Banco de Dados.** Interage com aplicações e consultas enviadas pelo usuário. Aceita consultas e examina esquemas externos e conceituais para determinar quais registros conceituais são requeridos para satisfazer uma consulta. Chama o gerenciador de arquivos para atender à requisição. Possui os seguintes componentes:
 - **Controle de Autorização.** Verifica se o usuário possui a autorização necessária para executar a operação requerida
 - **Processador de Comandos.** Responsável por controlar a operação quando se sabe que o usuário possui a autorização necessária
 - **Validador de Integridade.** Verifica se a operação requisitada satisfaz as restrições de integridade necessárias (como as restrições de chaves)
 - **Otimizador de Consultas.** Determina a estratégia para execução otimizada de consultas
 - **Gerenciador de Transações.** Executa o processamento das operações requeridas. Recebe as validações de integridade das transações
 - **Agendador.** Garante que operações concorrentes no banco de dados procedam sem conflitos entre si. Controla a ordem relativa em que as transações são executadas
 - **Gerenciador de Recuperação.** Garante que o banco de dados permaneça num estado consistente quando uma falha ocorrer (como violação de restrições). É responsável por confirmar ou abortar uma transação
 - **Gerente de Buffer.** É responsável por transferir dados entre a memória principal e o disco
- **Gerenciador de Arquivos.** Manipula os arquivos internos de armazenamento e gerencia a alocação de espaço em disco. Não é responsável por gerenciar diretamente entradas e saídas de dados. Ao contrário, passa a requisição para o método de acesso apropriado
- **Processador DML.** Converte as instruções da Linguagem de Manipulação de Dados encontradas nas aplicações em chamadas de funções do banco de dados. Interage com o processador de consultas para gerar o código apropriado.
- **Compilador DDL.** Converte as instruções da Linguagem de Definição dos Dados em um conjunto de tabelas que contém os meta-dados. As tabelas são armazenadas no catálogo de sistema enquanto que informações de controle são armazenadas no cabeçalho dos arquivos de dados
- **Gerenciador de Catálogo.** Gerencia acesso e mantém os catálogos de sistema

¹ Codd E. F. (1982) *The 1981 ACM Turing Award Lecture: Relational Database: A Practical Foundation for Productivity*. Comm. ACM, 25(2), 109-117

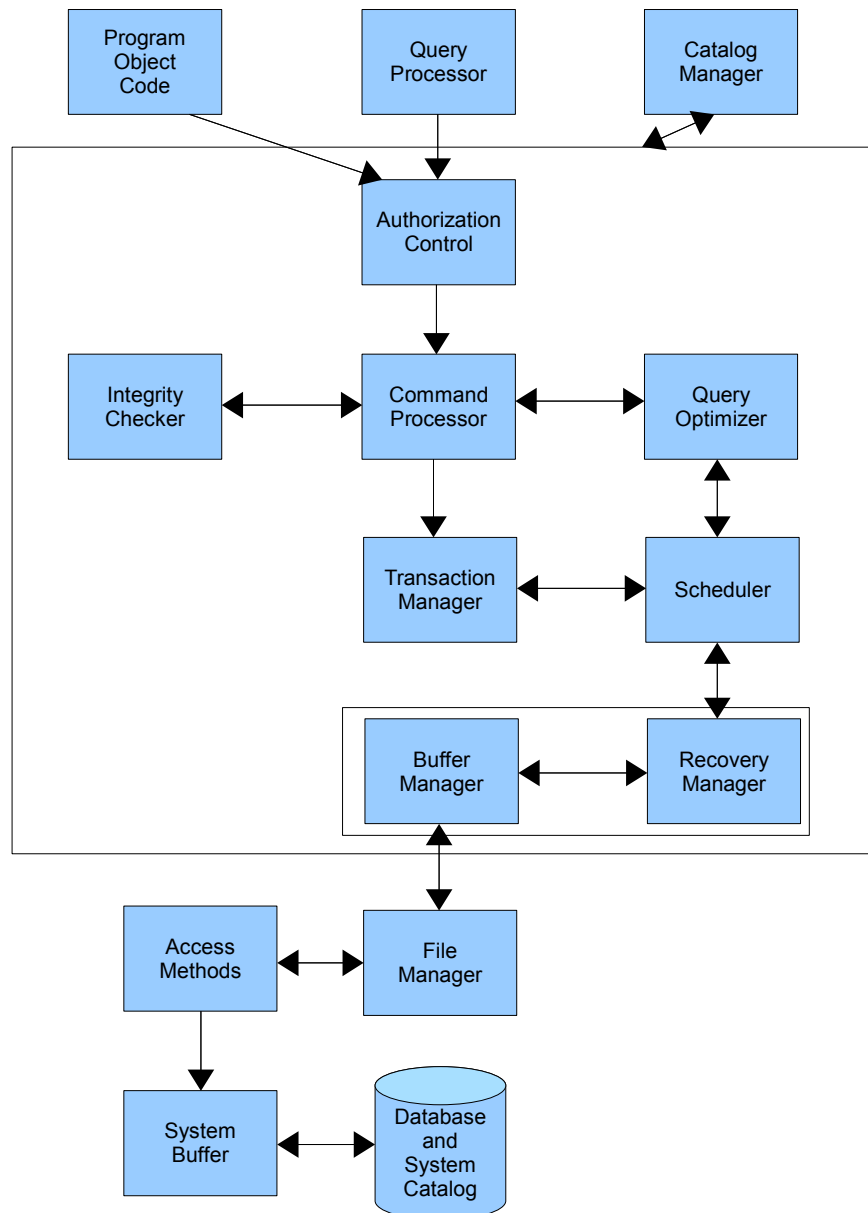


Figura 6: Arquitetura de Banco de Dados

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.

Módulo 9

Banco de Dados



Lição 2

Modelo Entidade-Relacionamento

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

1. Objetivos

O **modelo de entidade-relacionamento** é uma técnica de definição das necessidades de informação de uma organização. Provê alicerces para produtos de alta qualidade, adequando os sistemas às regras de negócio. Isto envolve identificar os objetos de importância para a organização (**entidades**), as propriedades desses objetos (**atributos**) e como eles se relacionam (**relacionamentos**). Seus dois objetivos principais são um modelo exato da necessidade de informação para a organização, que servirá como uma estrutura de desenvolvimento de novos sistemas, e prover um modelo independente de plataforma de armazenamento e formas de acesso, de modo que possam ser tomadas decisões técnicas de implementação e de coexistência com sistemas já desenvolvidos.

Ao final desta lição, o estudante será capaz de:

- Entender os conceitos básicos associados com o Modelo de Entidade-Relacionamento (MER) que descreve o modelo conceitual de alto nível o qual fornece suporte ao projeto do banco de dados
- Aplicar a técnica de diagramação dos modelos de entidade-relacionamento para representar situações comuns de negócio

2. Modelo Entidade-Relacionamento

O Modelo Entidade-Relacionamento é uma representação lógica detalhada dos dados de uma organização ou de uma área de negócio. É apresentado em termos de entidades no ambiente de negócio, seus relacionamentos e atributos. Este conceito foi introduzido por **Chen** em 1976.

2.1. Área de Atuação

Para organizar o modelo conceitual, utilizamos o conceito de Área Temática. **Área Temática** é uma área de interesse da empresa voltada para os recursos principais, produtos, atividades e sobre os quais devem ser mantidas informações. Isto pode ser feito pelo agrupando de entidades. Normalmente, usam-se substantivos plurais para representar uma Área Temática. Um exemplo de uma Área Temática é mostrado na seguinte figura:

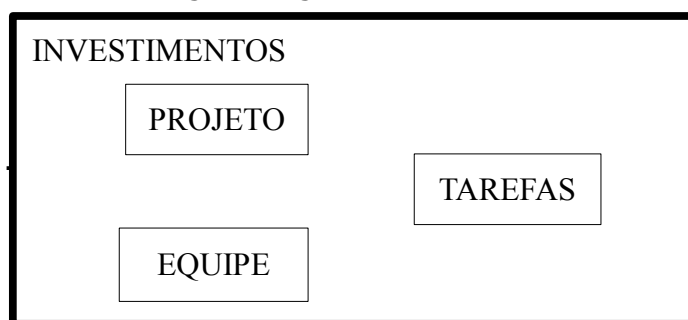


Figura 1: Exemplo de Área Temática

Nesse exemplo, a Área Temática INVESTIMENTOS consiste de três entidades, nomeadas PROJETO, TAREFA e EQUIPE. Os propósitos de existirem Áreas Temáticas são prover estrutura necessária para ajudar a focar nas especificidades do negócio e definir o escopo do sistema.

2.2. Entidades

Entidades podem ser pessoas, lugares, objetos, eventos ou conceitos de ambiente do usuário aos quais a organização precisa manter dados. Seguem exemplo de entidades:

Pessoa: EMPREGADO, ESTUDANTE, PACIENTE

Lugar: ESTADO, REGIAO, PAIS

Objeto: MAQUINA, PREDIO, AUTOMOVEL

Tipos de Entidades são coleções de entidades que compartilham propriedades ou características em comum. Também são conhecidas como **Classe de Entidade** e também são representadas por uma caixa retangular com o nome da entidade no seu interior. O nome deve ser em letras maiúsculas e no singular.

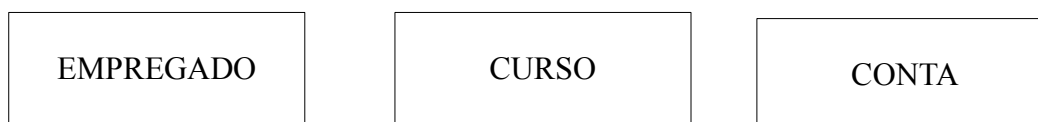


Figura 2: Exemplos de Entidades

Uma **instância de entidade**, por outro lado, é uma ocorrência única de um tipo de entidade. Um exemplo é mostrado na seguinte figura:

Tipo de Entidade:	EMPREGADO	Instâncias de EMPREGADO
Atributos:		
NUMERO EMPREGADO		642-17-8630
NOME		Michelle Brady
ENDERECO		100 Pacific ave.
CIDADE		San Francisco
ESTADO		Ca
CEP	98713	
ANO ADMISSAO		1989

2.3. Atributos

São propriedades ou características de uma entidade que são de interesse para a organização. São representados como elipses e são conectados por uma linha à entidade associada. Seus nomes são expressos usando-se letras maiúsculas. Como exemplo, considere a entidade ESTUDANTE com os seguintes atributos:

NUMERO ESTUDANTE
PRIMEIRO NOME
ULTIMO NOME
NOME DO MEIO
ENDERECO
TELEFONE

A entidade ESTUDANTE é modelada na seguinte figura:

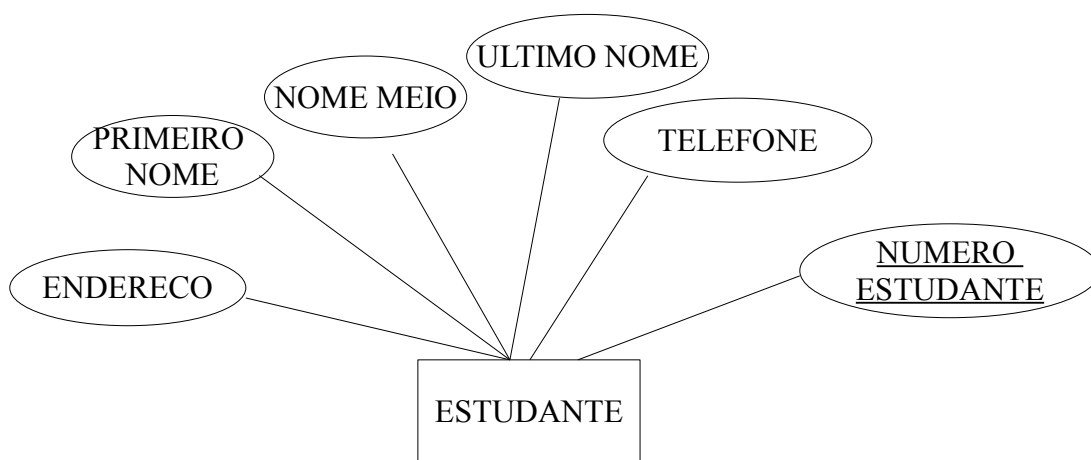


Figura 4: Entidade Estudante com atributos

Uma **chave candidata** é um atributo ou combinação de atributos que identificam unicamente cada instância de uma entidade. Significa que duas instâncias não podem ter o mesmo valor para estes atributos. Uma **chave composta** é uma chave candidata que contém mais de um atributo.

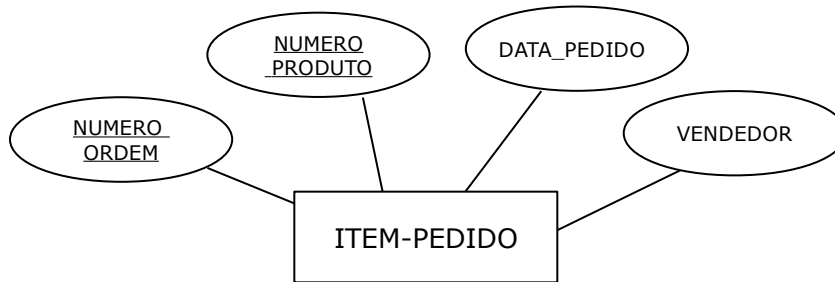
Uma **chave primária** é uma chave candidata que pode ser selecionada como um identificador único para uma entidade. Não pode conter valores nulos (NULL¹). É representada no Modelo Entidade-Relacionamento com seu nome sublinhado. Na figura 4, NUMERO ESTUDANTE é a chave primária da entidade ESTUDANTE, isto significa que dois estudantes não podem ter o mesmo número.

O critério para selecionar as chaves primárias são as seguintes:

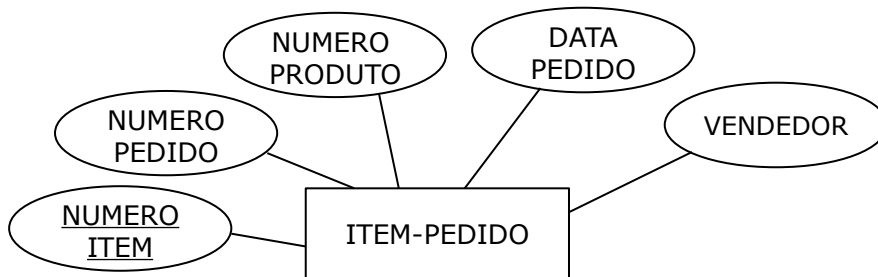
1. Escolher a chave candidata que não terá seu valor alterado no decorrer da vida de cada instância
2. Escolher a chave candidata que, para cada instância da entidade, tenha garantidamente valores válidos e não nulos. Se a chave candidata é uma combinação de dois ou mais atributos, tenha certeza que todas as partes da chave terão valores válidos, não nulos
3. Evitar o uso das chamadas "chaves inteligentes", cujas estruturas indicam classificação, localização, entre outros. Exemplo: Numero de Empregado é MA-10001 onde MA significa Matriz da empresa. Deve-se evitar isso porque quando o registro for modificado, dificultará modificar a chave primária assim como quando o empregado mudar de localização.
4. Considere substituir longas chaves compostas por atributos únicos. Veja na Figura 5, a entidade ITEM-PEDIDO tem inicialmente NUMERO PEDIDO, NUMERO PRODUTO e DATA PEDIDO como a chave primária. NUMERO ITEM tornou-se a chave primária, substituindo a

¹ Null não é zero (0), string vazio ou branco; significa um valor inexistente.

chave composta.



a) ITEM-PEDIDO tem chave composta de NUMERO PEDIDO, NUMERO_PRODUTO e DATA_PEDIDO.



b) ITEM-PEDIDO usa NUMERO_ITEM como a chave primária em detrimento da chave composta NUMERO_PEDIDO, NUMERO_PRODUTO, e DATA_PEDIDO.

Figura 5: Chave primária substituta

Chaves primárias podem ser:

- Atribuídas por sistema, automaticamente geradas por sistema, não necessariamente geradas por computador como um número da fatura ou um número oficial.
- Informadas pelo usuário, como um código de departamento.

Chave estrangeira é um atributo o grupo de atributos que é a chave primária de outra entidade. Ela é representada no modelo com o nome sublinhado por uma linha pontilhada.

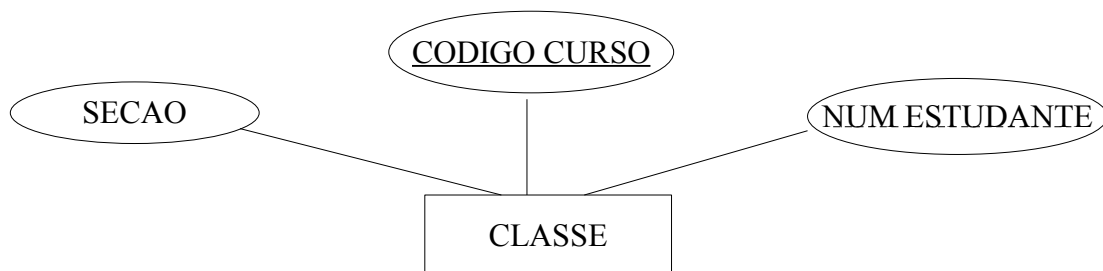


Figura 6: Ilustração de atributos

Na figura mostrada, a entidade CLASSE possui três atributos, chamados, SECAO, CODIGO CURSO e NUM ESTUDANTE. CODIGO CURSO é a chave primária enquanto NUM ESTUDANTE é uma chave estrangeira (que é a chave primária da entidade ESTUDANTE).

Existem atributos cujos valores são computados ou derivados de outros atributos existentes. Estes atributos são conhecidos como **dados derivados**. Não é recomendado utilizar tais atributos, pois causam dados redundantes que dificultam a atualização. Entretanto, existem casos em que a performance pode ser melhorada criando esses tipos de dados. Exemplo desses tipos de dados são CONTADORES, TOTALIZADORES e MÉDIAS.

Atributos multivalorados são atributos que podem conter mais de um valor para cada instância de entidade. Um exemplo de atributo multivalorado é mostrado na figura a seguir.

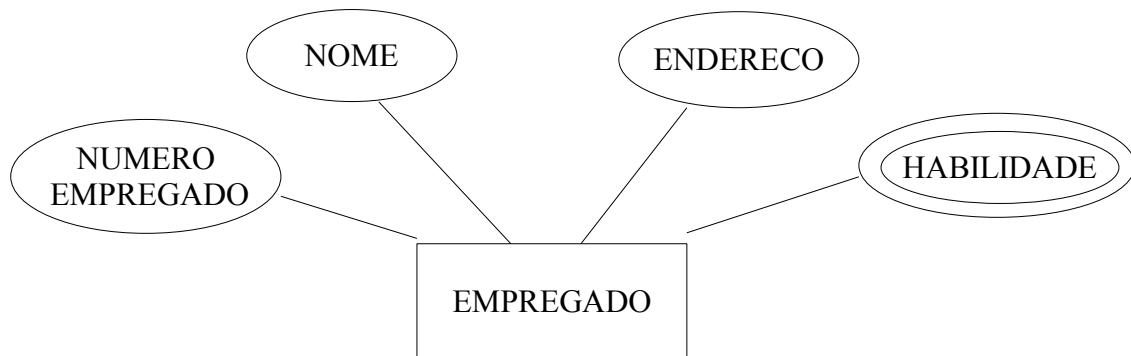


Figura 7: Atributos multi valorados

Nesse exemplo, um EMPREGADO pode ter várias ocorrências de HABILIDADE.

2.4. Relacionamentos

Relacionamentos em um Modelo Entidade-Relacionamento são associações entre instâncias de um ou mais tipos de entidades que são de interesse da organização. São considerados como a cola que une os vários componentes de Modelo Entidade-Relacionamento. Podem ser expressos de duas formas:

1. Usar verbos dentro de um losango
2. Usar verbos colocados nas linhas que conectam as entidades

Pode-se usar duas convenções. Entretanto, deve-se manter a consistência. Se a primeira forma de expressar relacionamento é utilizada, continue assim. As duas formas de representar relacionamentos são mostradas na figura a seguir. O relacionamento é lido como "Um EMPREGADO freqüentou um CURSO, um CURSO é freqüentado por um EMPREGADO".

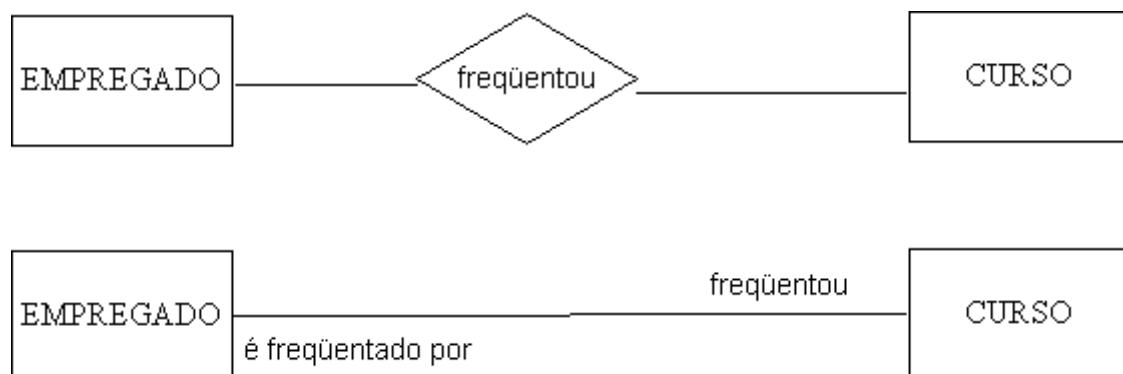


Figura 8: Duas formas de expressar um relacionamento

O grau de um relacionamento é a quantidade de tipos entidades que participam de um relacionamento. O três graus mais comuns de relacionamentos são unário, binário e ternário.

Um relacionamento unário ocorre um relacionamento entre instâncias da mesma entidade. É também conhecida como relacionamento recursivo. A figura 9 mostra como modelar relacionamentos unários. Nesse exemplo uma PESSOA só pode ser casada com outra PESSOA. Outro exemplo é a figura 10 onde um EMPREGADO (gerente) pode gerenciar um ou mais EMPREGADOS; e, um EMPREGADO é gerenciado por outro EMPREGADO (gerente). O último exemplo mostrado na figura 11 demonstra que um ITEM consiste de vários componentes; um ITEM pode ser parte de vários componentes. Nesse exemplo, o relacionamento pode ter um atributo.

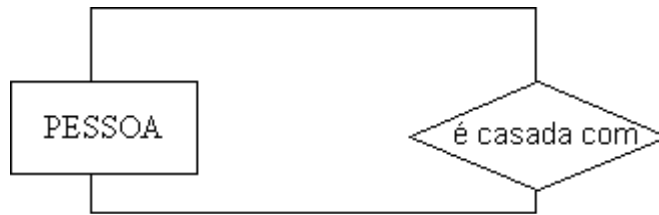


Figura 9: Relacionamento unário (Um-para-um)

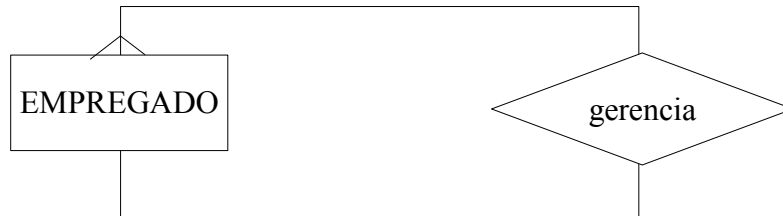


Figura 10: Relacionamento Unário (um-para-muitos)

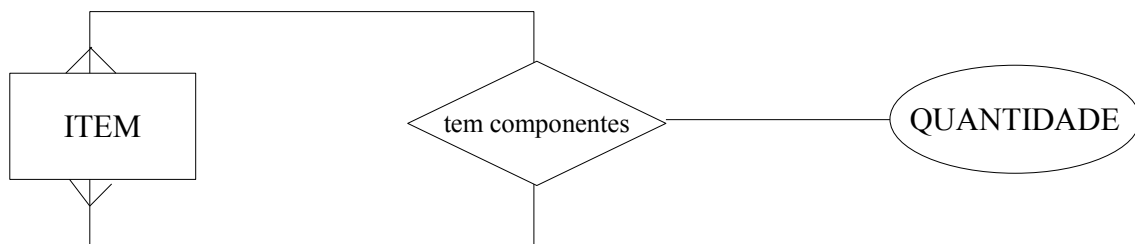


Figura 11: Relacionamento Unário (muitos-para-muitos)

Um **relacionamento binário** ocorre entre instâncias de dois tipos de entidade. Este é o tipo de relacionamento mais comum. Na figura a seguir, a um EMPREGADO é atribuído a uma vaga de garagem. Uma VAGA DE GARAGEM é designada para um EMPREGADO.

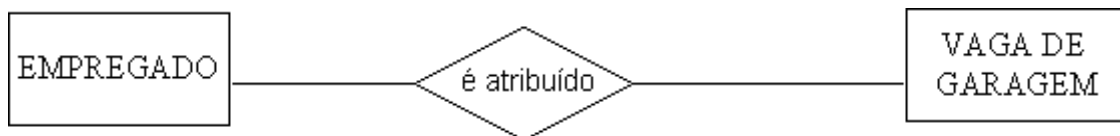


Figura 12: Relacionamento Binário (Um-para-um)

Na figura a seguir, uma LINHA DE PRODUTO contém muitos PRODUTOS. Um PRODUTO está contido em uma LINHA DE PRODUTO. Este é um exemplo de um relacionamento binário com cardinalidade um-para-muitos.

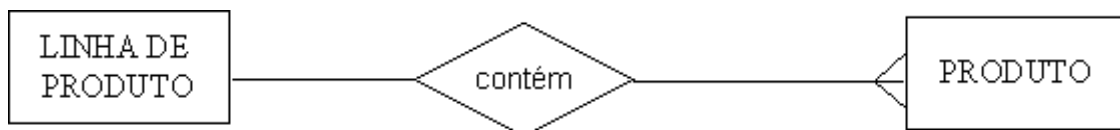


Figura 13: Relacionamento binário(um-para-muitos)

Na figura a seguir, um ESTUDANTE pode se matricular em um ou muitos CURSOS; um CURSO pode ter um ou muitos ESTUDANTES matriculados. Este é um exemplo de relacionamento binário com cardinalidade muitos-para-muitos.

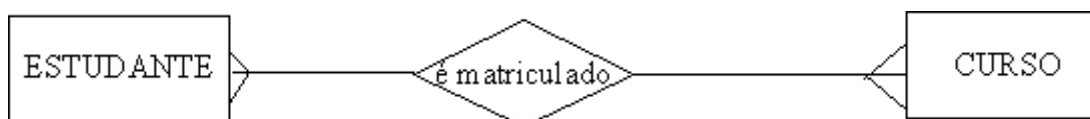


Figura 14: Relacionamento binário (Muitos-para-muitos)

O **relacionamento ternário** é um relacionamento simultâneo entre instâncias de três tipos de entidades. A figura a seguir mostra um exemplo. Muitas PARTES são expedidas de diversos POSTOS de diferentes FORNECEDORES.

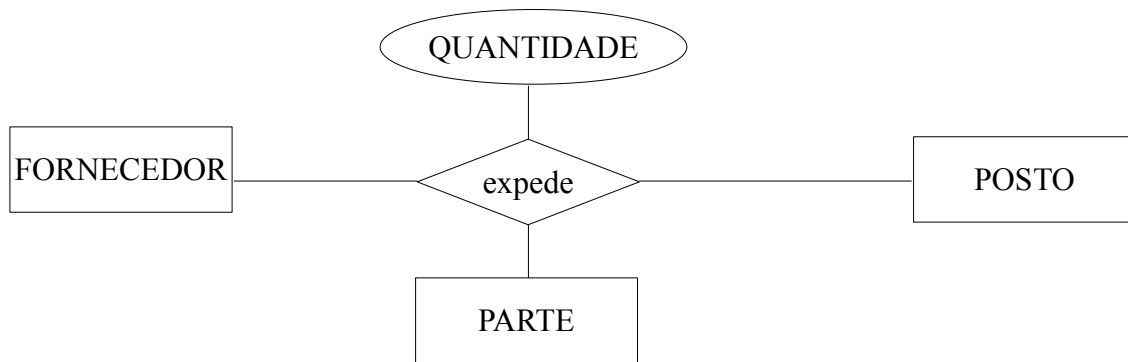


Figura 15: Relacionamento Ternário

A **cardinalidade de um relacionamento** é o número de instâncias da entidade B que pode ser ou deve ser associada com cada instância da entidade A. Partindo do princípio de que a entidade A está relacionada com a entidade B, são modelados como segue:

1. **Cardinalidade obrigatoriamente um** é modelada como duas linhas verticais próximo ao símbolo da entidade. É lido como "um e somente um".

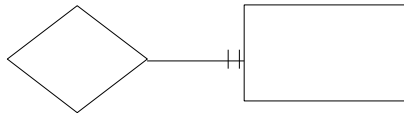


Figura 16: Cardinalidade um mandatória

2. **Cardinalidade múltipla** (1, 2, ..., muitos) é modelada como um pé de galinha. Lida como "um para muitos". Demonstrado na figura a seguir.

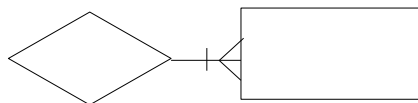


Figura 17: Cardinalidade Múltipla

3. **Cardinalidade opcional** (0 ou 1) é modelada com um círculo e uma linha vertical perto da entidade. É lida como "zero ou um".

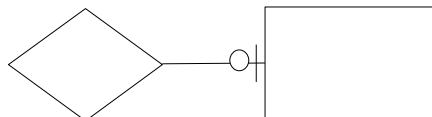


Figura 18: Cardinalidade opcional 0 ou 1

4. **Cardinalidade opcional** (0-muitos) é modelada como um círculo e um pé de galinha perto da entidade. É lida como "zero ou muitos".

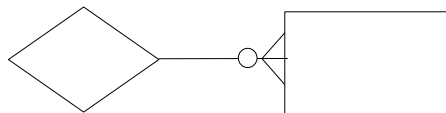


Figura 19: Cardinalidade opcional zero ou muitos

A **cardinalidade mínima** de um relacionamento é o número mínimo de instâncias da entidade B que podem ser associadas a cada instância da entidade A. A **cardinalidade máxima** de um relacionamento é o número máximo de instâncias que podem ser associadas a cada instância da entidade. Na Figura 20, uma PRATELEIRA pode ter zero ou mais PRODUTOS; um PRODUTO é estocado em uma e só uma PRATELEIRA.

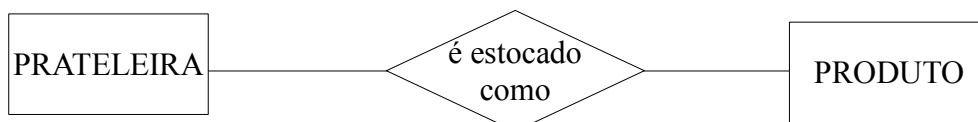


Figura 20: Cardinalidade do Relacionamento

Se a cardinalidade mínima é zero, a participação da entidade é opcional. Se a cardinalidade mínima é um, a participação é **obrigatória**.

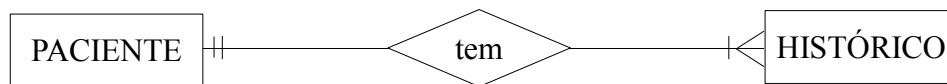


Figura 21: Ambos os lados tem participação obrigatória

Um exemplo de quando os dois lados do relacionamento têm participação obrigatória: um **PACIENTE** deve ter, pelo menos, um **HISTÓRICO**; e, um **HISTÓRICO** é associado com um e só um **PACIENTE**.

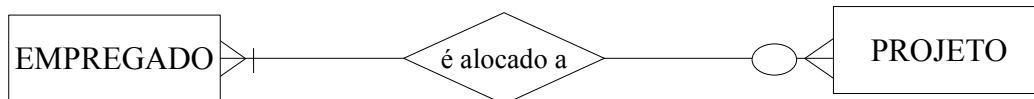


Figura 22: Um lado tem a participação obrigatória e o outro tem a participação opcional

Outro exemplo onde um lado tem participação obrigatória e o outro tem participação opcional. Um **EMPREGADO** pode ser alocado a um ou mais **PROJETOS**, ou não, um **PROJETO** é alocado a, pelo menos, um **EMPREGADO**.



Figura 23: Relacionamento unário com participação opcional

Esta figura mostra uma participação opcional em um relacionamento unário. Uma **PESSOA** é casada com uma e somente uma pessoa, ou não, neste caso, a pessoa é solteira.

Uma **dependência de existência** significa que uma instância de uma entidade não pode existir sem alguma outra entidade. **Entidade fraca** é um tipo de entidade que possui uma dependência de existência. Um **relacionamento identificado** é aquele onde a chave primária da entidade pai é usada como parte da chave primária da entidade dependente. A figura a seguir mostra um exemplo de um relacionamento identificado. **COPIA DE FILME** é uma entidade fraca porque tem uma dependência de existência com um **FILME**. Uma **COPIA DE FILME** não pode existir sem um **FILME**.

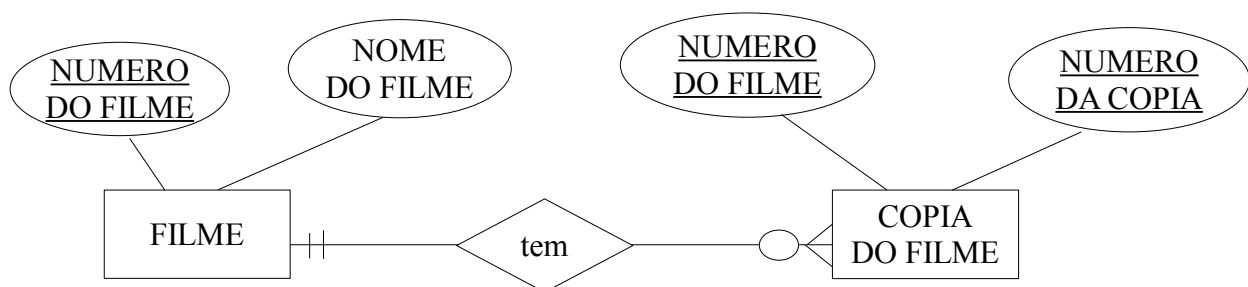


Figura 24: Dependências de Relacionamento

Alguns benefícios de se identificar relacionamentos:

1. integridade dos dados
2. facilidade de acesso dos dados dependentes

3. Análise de Situação: Descobrindo Entidades, Atributos e Relacionamentos

Uma maneira fácil para se descobrir entidades, atributos e relacionamentos consiste em perguntar ao usuário sobre suas perspectivas. Normalmente, a resposta do usuário é uma situação. Uma situação é um conjunto bem-definido de circunstâncias que pode ser descrito usando uma linguagem natural suficientemente completa. Uma linguagem natural completa inclui as seguintes construções gramaticais: substantivo, verbo e modificadores.

3.1. Entidades e Substantivos

Um **substantivo** é o nome de um tipo de pessoa, animal, planta, lugar, coisa, substância ou idéia. Também pode representar objetos animados ou inanimados que podem ser tangíveis ou intangíveis. É impossível de se especificar qualquer situação sem se usar, pelo menos, um substantivo.

Um **substantivo próprio** é o nome de uma ocorrência particular ou instância de um substantivo. Um **pronome** é uma palavra usada como substituta de um substantivo e se refere a um substantivo conhecido no contexto onde é usado.

Entidades em qualquer situação são substantivos.

Os passos para se descobrir entidades seguem:

- Identificar e definir a entidade
 - Questionar-se, "Qual pessoa, lugar ou objeto uma empresa quer guardar informações?"
 - Procurar pelo tipo ao invés das instâncias ou ocorrências. Instâncias estão na forma de substantivos próprios
 - Usar os tipos comuns como o nome da entidade
- Determinar a chave primária da entidade
 - Questionar-se, "O que diferencia unicamente as instâncias de uma entidade?"
 - Caso não exista nenhuma chave primária, criar uma coluna de código ou número da chave primária da entidade
- Documentar a entidade

Nem todos os substantivos são entidades (todas as entidades são substantivos). Alguns deles são atributos disfarçados. Entidades existem por si só. Atributos têm significados somente no contexto das entidades que os descrevem. Entidades são caracterizadas usando atributos e não possuem características próprias. Exemplos de atributos em disfarce são "nome", "descrição" e "cor".

3.2. Relacionamentos e Verbos

Um **verbo** é uma palavra que descreve um modo de ser, uma associação, uma ação ou um evento. Verbos descrevem o estado dos substantivos e os relacionam. Relacionamentos são verbos de uma situação.

Os passos para se descobrir relacionamentos seguem:

- Identificar e definir o relacionamento
 - Questionar-se, "Qual o relacionamento da <entidade A> com a <entidade B>?"
- Determinar o grau do relacionamento
 - Questionar-se, "É unário, binário ou ternário?"
- Determinar a cardinalidade do relacionamento

- Questionar-se, "É obrigatoriamente um, muitos, opcional 0-1 ou opcional zero-muitos?"
- Documentar o relacionamento

3.3. Atributos e Modificadores

Um **modificador** é uma palavra que qualifica um substantivo ou um verbo sobre seu caráter, quantidade ou extensão. Um **adjetivo** é um modificador de um substantivo. Um **advérbio** é um modificador de um verbo. Atributos são modificadores de uma situação.

Os passos para se descobrir atributos seguem:

- Identificar e definir o atributo
 - Questionar-se, "Quais características de uma entidade ou relacionamento as quais interessam à organização?"
- Determinar quais atributos são realmente atributos, e quais são entidades.
 - Verificar a frase preposicional

Por exemplo: Salário do Empregado, Nome do Cliente

Salário somente é definido dentro do contexto de um empregado. Similarmente, nome é definido dentro do contexto de um cliente. Salário e nome são atributos do empregado e do cliente, respectivamente

- Realizar o processo de normalização. Normalização é o processo de converter estruturas complexas em estruturas simples e estáveis. Embora esse processo seja feito normalmente na fase de projeto do banco de dados, pode ser usado para simplificar formas que os usuários mostram durante a fase de análise do desenvolvimento de software. A normalização é discutida com mais detalhes no capítulo do Projeto Lógico de Banco de Dados
- Usar da intuição
- Documentar o atributo

Considere o seguinte pedaço de entrevista para uma Loja Orgânica:

Entrevistador: Quais pessoas, lugares ou objetos deseja manter como registro?

Entrevistado: "Bem, Ma, eu e os outros 300 empregados temos administrado essa pequena loja de alimentos saudáveis muito bem sem utilizar computadores, entretanto reconhecemos que poderíamos ter um maior controle, entende o que eu digo? Claro que sim. Podemos criar Códigos de Empregados se for necessário..."

"De qualquer maneira, é necessário termos os registros de todos os diferentes tipos de itens - germen de trigo, vitamina E, granola coberta com chocolate - este vende muito bem. Séria melhor bolarmos um número para todos, pois são muitos tipos de itens. E precisamos saber quais itens são vendidos e em quais lojas e quais não. Seria interessante conhecer quanto cada loja tem em estoque. Não temos números para as lojas, mas podemos criá-los..."

"Temos lojas em 3 estados do País? E estamos pensando em abrir novas filiais em outros estados..."

Texto 1: Exemplo da entrevista de uma Loja Orgânica

O primeiro passo é identificar as entidades. Identificar os substantivos no texto. Alguns exemplos estão listados abaixo:

Empregado
Loja
Computador
Número do Empregado

Germe de Trigo, Vitamina E e Granola Coberta com Chocolate
Estados

Nem todos os nomes são entidades; alguns são atributos disfarçados. Alguns exemplos são:

1. NUMERO DE EMPREGADO tem significado no contexto do EMPREGADO, por exemplo, é um atributo de um EMPREGADO.
2. Considere a sentença. "Melhor você dar número a todos eles, porque nós temos muitos tipos de itens." Os NUMEROS mencionados são um atributo da entidade ITEM. E o NUMERO também é um atributo da LOJA na sentença, "Não temos números para as lojas, mas podemos criá-los..."
3. Considere a enumeração, "...germe de trigo, vitamina E, granola coberta com chocolate...". Representam exemplos de itens sendo vendidos. O substantivo comum que representaria tudo isso seria ITEM ou PRODUTO. No modelo, ITEM é o nome da entidade.

A figura abaixo mostra o diagrama inicial das entidades que foram descobertas até agora.

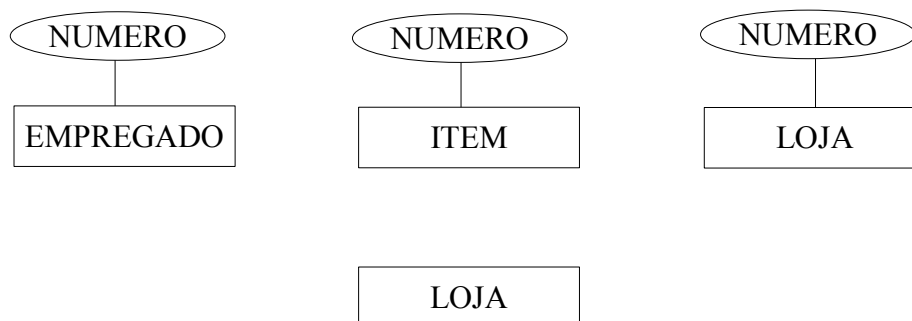


Figura 25: Entidades Iniciais

O próximo passo é identificar os relacionamentos entre as entidades. Nesse caso, os verbos são analisados para checar se há uma conexão entre as entidades. Alguns exemplos são:

1. Considere a sentença, "...precisamos saber quais itens são vendidos e em quais lojas e quais não...". O verbo "vendido" conecta ITENS às LOJAS.
2. Considere a sentença, "...temos lojas em 3 estados do país?...". O verbo "temos" conecta as LOJAS ao ESTADO.
3. Em alguns casos, relacionamentos não são explicitamente mostrados, e precisam ser descobertos pela **intuição**. Por exemplo, EMPREGADOS trabalham em LOJAS.

Assim que os relacionamentos forem identificados, a cardinalidade dos relacionamentos deve ser definidas.

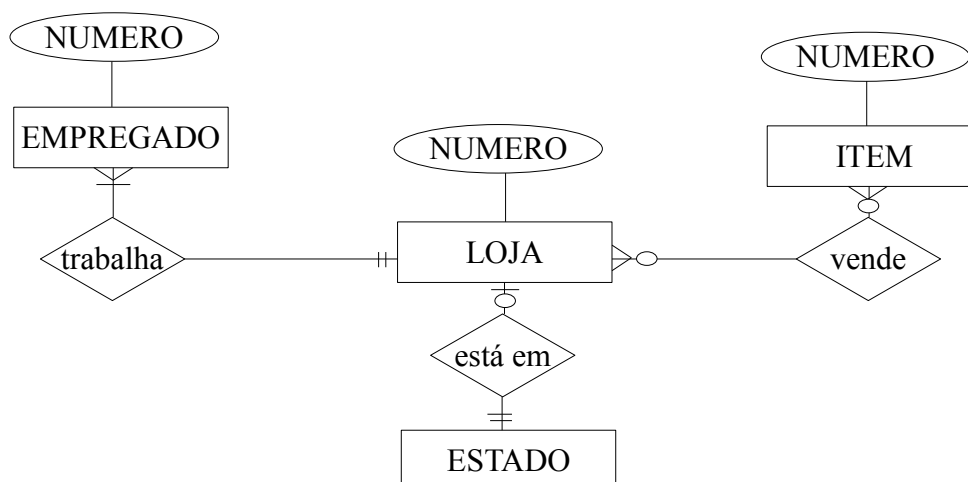


Figura 26: Entidades e seus Relacionamentos definidos

A figura mostra a modelagem dos relacionamentos e suas cardinalidades em um primeiro estágio.

Para finalizar, identificamos atributos adicionais olhando os adjetivos, advérbios e frases preposicionais. Alguns exemplos:

1. Considere a frase, "...Seria interessante conhecer quanto cada loja tem em estoque...". Nesse exemplo, querem saber quantos itens existem em cada loja. Isso é um atributo do relacionamento "vende".
2. Em alguns casos, use o senso comum para manter os registros como nome dos empregados, loja e estado.

Identifique a chave primária da entidade. A figura a seguir mostra o MER do transcrito da entrevista.

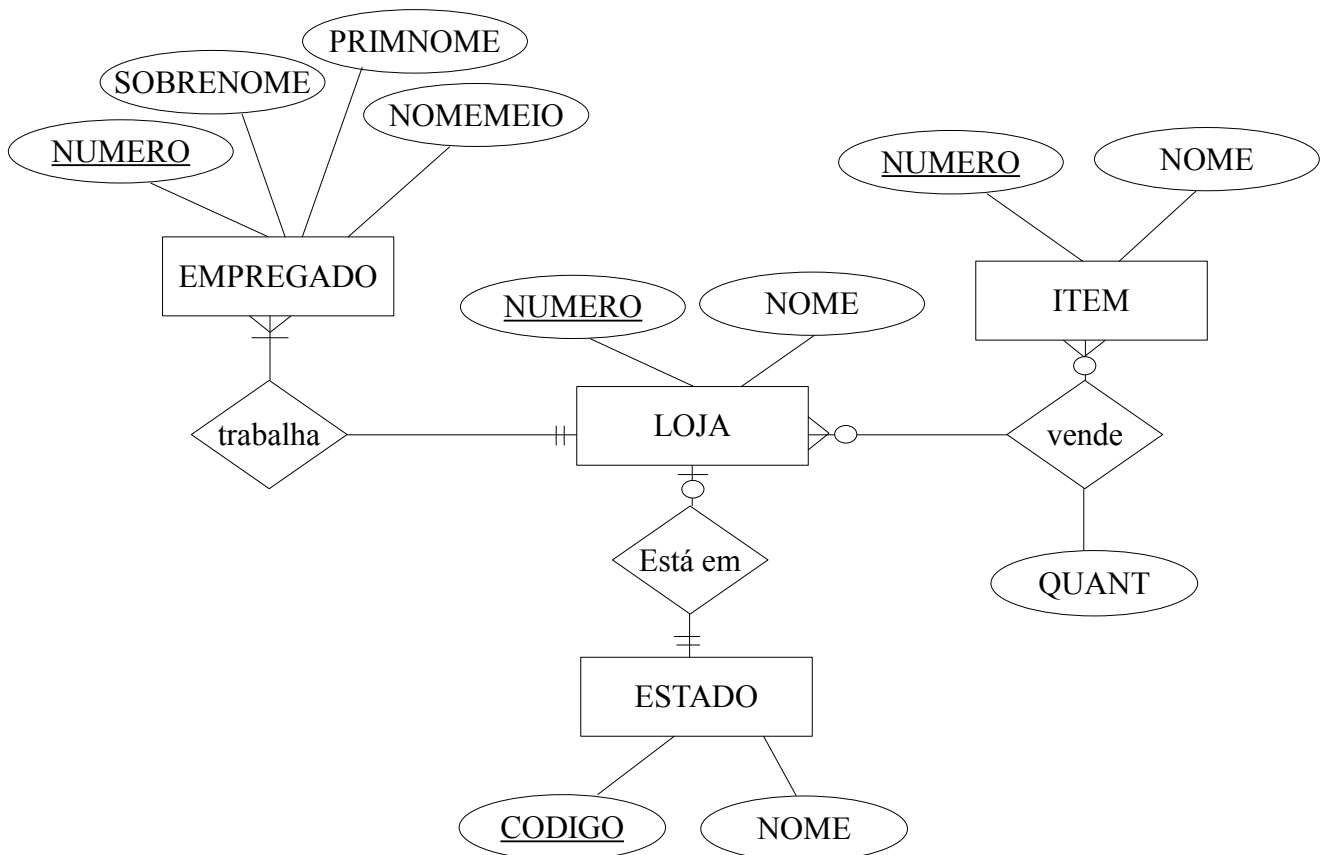


Figura 27: Modelo Entidade-Relacionamento para Loja Orgânica

3.4. Modelo de Entidade-Relacionamento Avançado

Gerúndios

Gerúndios são relacionamentos **muitos pra muitos** que o administrador de dados escolhe modelar como tipos de entidade com vários relacionamentos de **um pra muitos**. Como exemplo, considere a seguinte figura que mostra ENTREGA é modelado como um gerúndio.

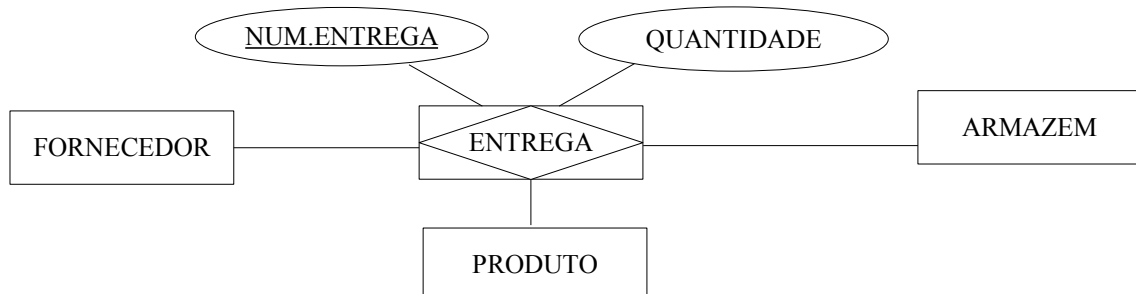


Figura 28: Exemplo de Gerúndio

Vários FORNECEDORES entregam vários PRODUTOS para vários ARMAZENS. NUMERO DA ENTREGA e QUANTIDADE são atributos do gerúndio.

Modelando Atributos Multivalorados

É um atributo que pode conter mais de um valor. Cada atributo multivalorado, ou mais, geralmente cada grupo de repetição, é convertido em um tipo de entidade separada que tem um relacionamento com o tipo da entidade da qual foi removido. Considere a figura 29 com o atributo multivalorado chamado COMPETENCIA. Nesse exemplo, um empregado tem muitas competências. Removemos o atributo multivalorado e o transformamos em uma entidade separada com um relacionamento com a entidade original. O MER modificado é mostrado na figura 30.

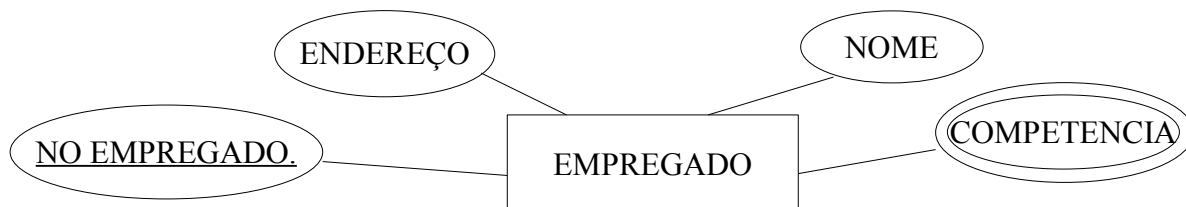


Figura 29: Empregado com atributo multivalorado

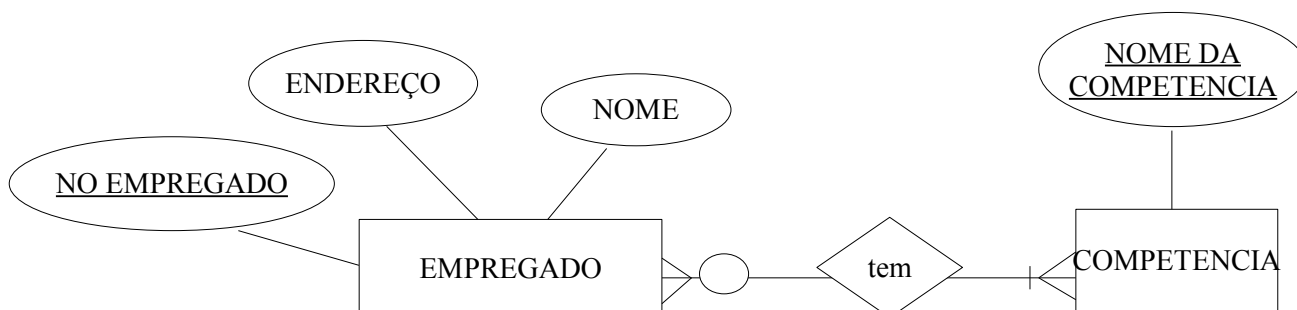


Figura 30: Atributo Multivalorado Remodelado

Modelando Grupos de Repetição

Um **grupo de repetição** é um conjunto de dois ou mais atributos multivalorados que estão logicamente relacionados. Considere um prontuário de um paciente como o mostrado na figura abaixo.

PATIENT CHART		
Número do Paciente: 639147-0		
Nome do Paciente: Michael Smith		
Endereço: 329 Fourth St.		
Data de		
<u>Visita</u>	<u>Médico</u>	<u>Sintoma</u>
24/03/94	Ryan	Febre
06/07/94	Nelson	Sore Garganta
16/08/94	Ryan	Resfriado
...

Figura 31: Exemplo de um Prontuário de Paciente

No exemplo mostrado acima, Data de Visita, Médico e Sintoma ocorrem várias vezes. Isto é um grupo de repetição que mostra o histórico do paciente. O MER (Modelo de Entidade-Relacionamento) deste prontuário é mostrado na figura abaixo.

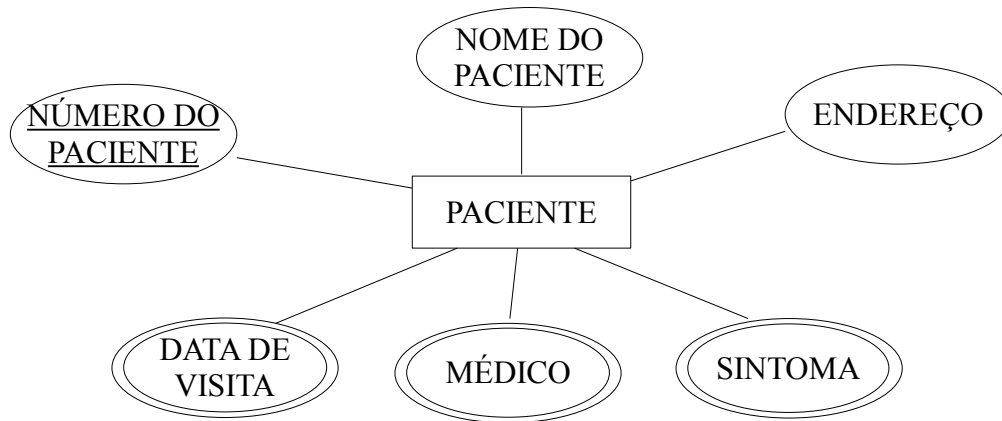


Figura 32: Grupo Repetido do Histórico do Paciente

O grupo repetido é removido através da criação de outra entidade que representaria este grupo. Uma chave primária pode ser criada ou escolhida entre os atributos existentes. A chave primária da entidade original torna-se uma chave estrangeira da nova entidade. A figura abaixo mostra o prontuário remodelado do paciente. Uma nova entidade chamada HISTÓRICO DO PACIENTE é criada e uma nova chave primária foi definida com o nome NÚMERO DO HISTÓRICO. O HISTÓRICO DO PACIENTE tem NÚMERO DO PACIENTE como sua chave estrangeira.

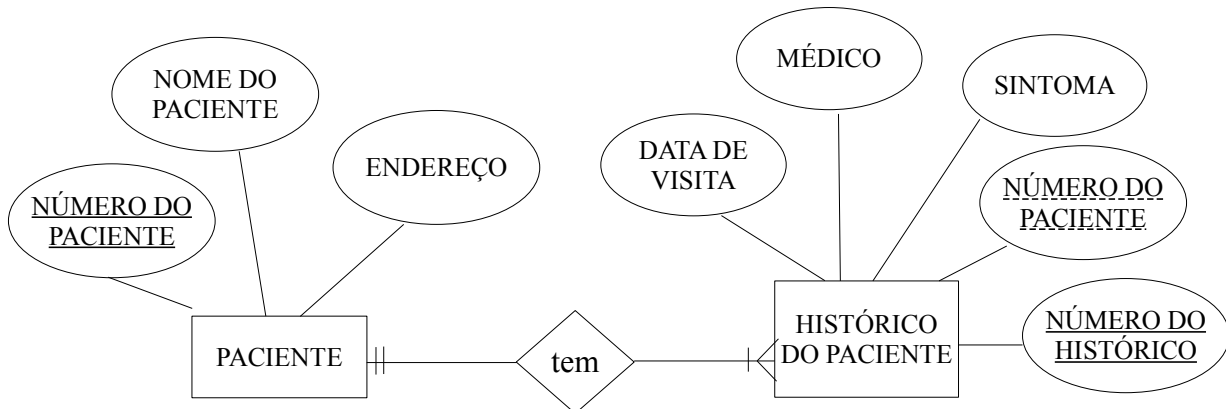


Figura 33: DER do prontuário do Paciente Remodelado

Modelando Dados dependentes de Tempo

Um selo de tempo (*time-stamp*) é simplesmente um valor de tempo (tal como data e hora) que é associado a um valor de data. Como exemplo, considere um produto cujo preço varie ao longo do tempo. Alterações de preço devem ser acompanhadas. A figura 34 mostra o MER deste exemplo. PREÇO e DATA EFETIVA são modelados como um grupo de repetição.

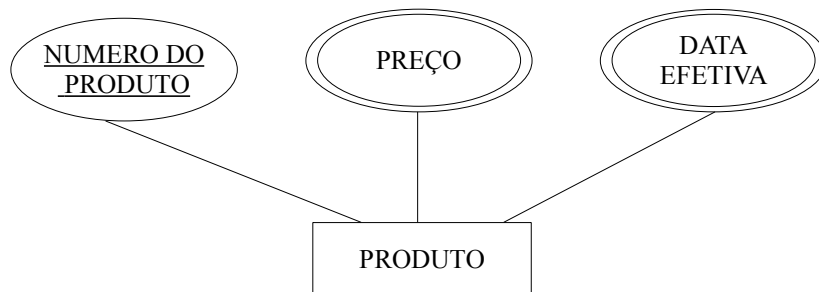


Figura 34: DER Preço varia com o tempo como um grupo de repetição

O grupo de repetição deve ser removido e uma nova entidade, criada. Uma nova entidade chamada HISTÓRICO DE PREÇO é criada com uma chave composta de PRODUTO NÚMERO e DATA EFETIVA. A figura abaixo mostra o MER remodelado do preço.

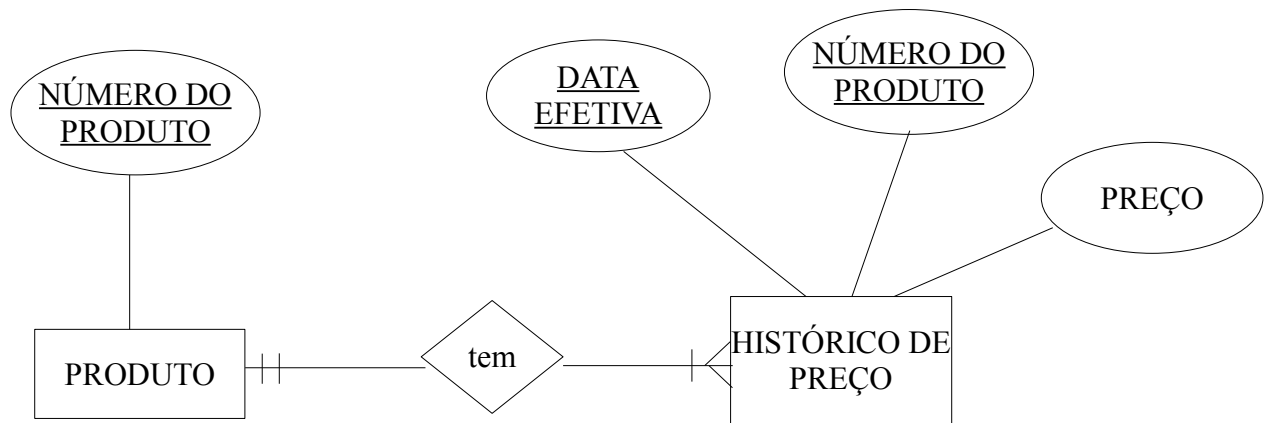


Figura 35: Mudança de Preço ao longo do tempo remodelado

Generalização e Categorização

Generalização é o conceito de que algumas coisas são subtipos de outras coisas mais gerais. Como exemplo, um passageiro de uma companhia aérea é um subtipo de um passageiro.

Categorização é o conceito de que algumas coisas se transformam em vários subtipos. Como exemplo, um sorvete pode ter vários sabores.

Um outro exemplo. Um EMPREGADO pode ser remunerado por hora (EMPREGADO-HORA), por mês (EMPREGADO ASSALARIADO) ou CONSULTORES. A lista de seus atributos é a que se segue:

Tipo de Entidade: EMPREGADO POR HORA

Atributos: NÚMERO DO EMPREGADO
NOME
ENDEREÇO
DATA DE ADMISSÃO
PERCENTAGEM HORÁRIA

Tipo de Entidade: EMPREGADO ASSALARIADO

Atributos: NÚMERO DO EMPREGADO
NOME
ENDEREÇO
DATA DE ADMISSÃO
SALÁRIO ANUAL
OPÇÕES DE AÇÕES

Tipo de Entidade: CONSULTOR

Atributos: NÚMERO DO EMPREGADO
NOME
ENDEREÇO
DATA DE ADMISSÃO
NÚMERO DO CONTRATO
PERCENTUAL DIÁRIO

Observe que estas entidades compartilham atributos. Existem várias formas de desenvolver modelos conceituais de generalização e categorização. A seguir as opções:

1. Definir uma entidade única EMPREGADO para representar todos os tipos.
2. Definir uma entidade separada para cada tipo.
3. Definir uma entidade supertipo EMPREGADO e entidades subtipos EMPREGADO POR HORA, EMPREGADO ASSALARIADO E CONSULTORES.

Na figura abaixo o MER de EMPREGADO usando a terceira opção.

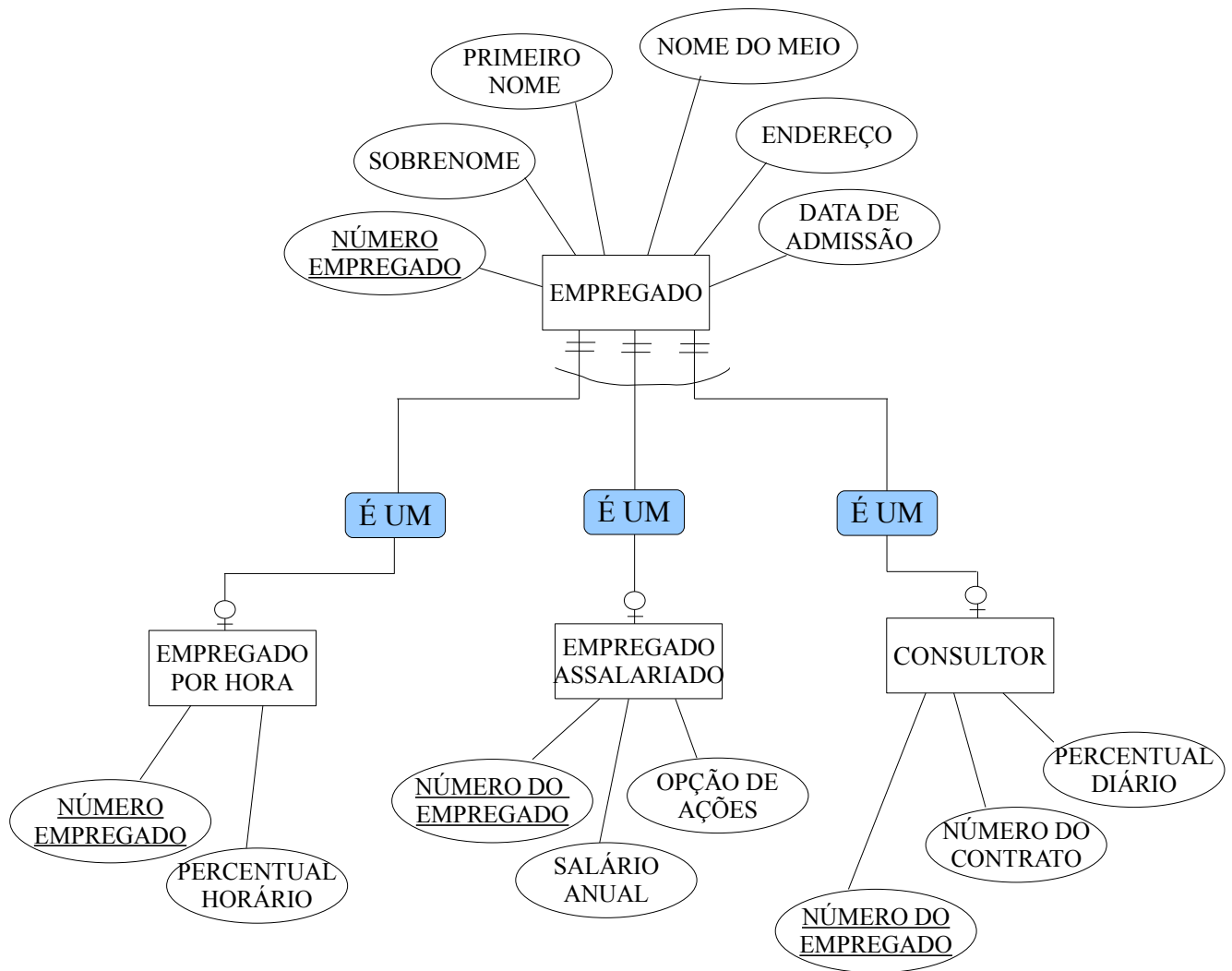


Figura 36: Modelo de Categorização de Empregado

Um **supertipo** é uma entidade genérica que é subdividida em subtipos. O supertipo contém atributos que são comuns a todos os seus subtipos. Um **subtipo** é um subconjunto de um supertipo. Contém atributos específicos de si próprio. A chave primária de subtipos é a mesma de uma chave primária do supertipo.

Um **relacionamento supertipo-subtipo** é um relacionamento chamado “é-um”. É representado com um retângulo com bordas arredondadas. A cardinalidade do relacionamento de um subtipo para um supertipo é sempre mandatória; enquanto que a cardinalidade do relacionamento de um supertipo para o subtipo é sempre opcional. A notação da cardinalidade é modelada opcionalmente.

Um relacionamento supertipo-subtipo é um **relacionamento exclusivo** quando os subtipos são mutuamente exclusivos e cada instância do supertipo é categorizada com exatamente um subtipo. É representada como uma linha curva logo abaixo da entidade supertipo.

Relacionamentos de **subtipos completos** ocorrem quando todos os subtipos são definidos por um supertipo. Não existem mais subtipos definidos para um supertipo em particular. O EMPREGADO e seus subtipos definidos a figura a seguir é um exemplo.

Relacionamentos de **subtipos não completos** ocorrem quando alguns (mas não todos) dos subtipos tenham sido definidos. Um exemplo é mostrado na figura a seguir. O supertipo VEÍCULO tem AUTOMÓVEL, CAMINHÃO e MOTO como seus subtipos. Contudo, pode haver outros tipos de veículos não definidos. O retângulo em branco representa a porção não completa do relacionamento.

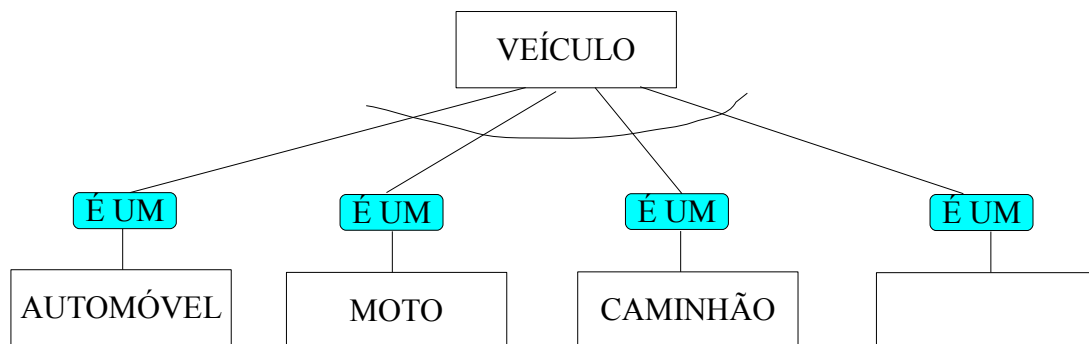


Figura 37: Relacionamentos de subtipos completos e não completos

Um Relacionamento de **subtipos não-exclusivos** ocorre quando subtipos podem se sobrepor. Uma instância de um supertipo pode simultaneamente pertencer a mais de um subtipo. A figura abaixo mostra um exemplo. Aqui, um veículo de direção 4x4 pode ser um Utilitário Esportivo ou Off-Road.

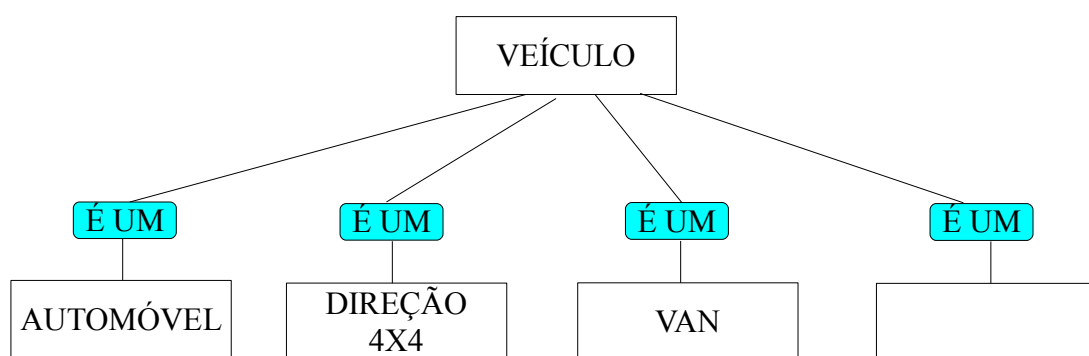


Figura 38: Subtipos não-exclusivo e não-exaustivo

Generalização ou categorização tem a propriedade chamada **herança**.

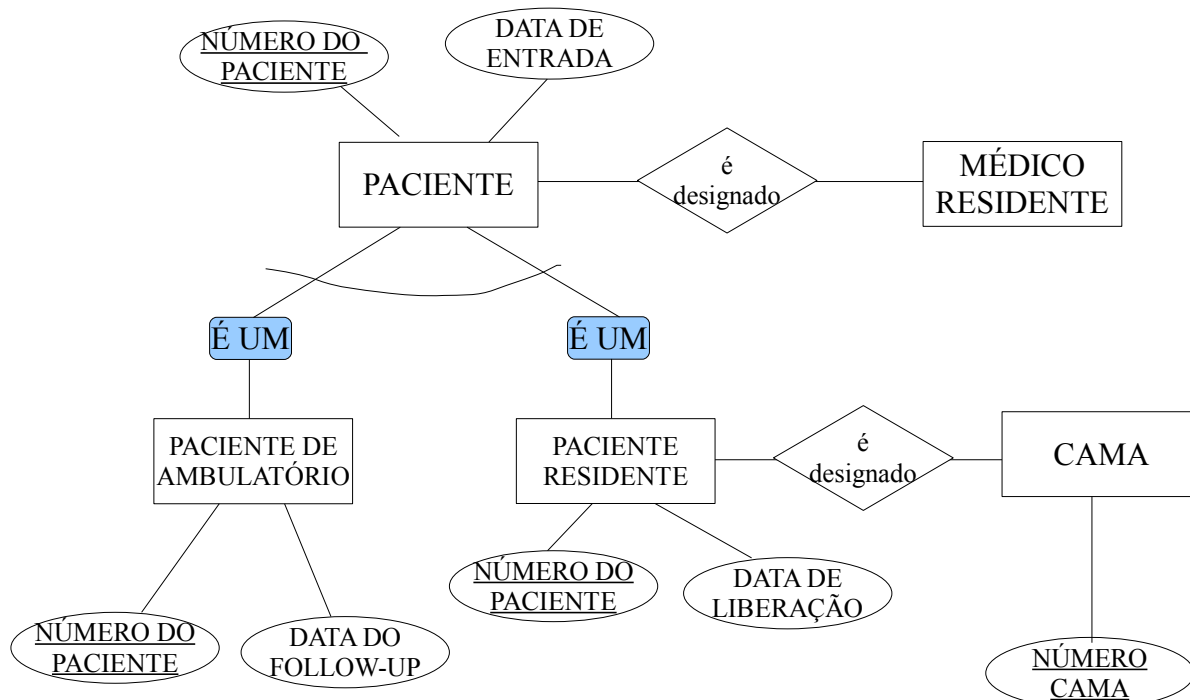


Figura 39: Exemplo de Herança

Significa que tipos de entidade ou classes de objeto são arranjados em uma hierarquia na qual cada tipo de entidade assume os atributos daqueles que estão mais acima na hierarquia. A figura acima ilustra um exemplo. Ambos PACIENTE DE AMBULATÓRIO e PACIENTE RESIDENTE herdam o atributo DATA DE ENTRADA e é atribuído o relacionamento da entidade PACIENTE.

4. Exercício

4.1. Entrevista 1

Leia a transcrição de uma entrevista feita com um Corretor de Imóveis.

O usuário diz:

"Tudo certo. Eles me disseram que você pode ajudar. E eu espero que você possa mesmo, pois, 1400 unidades, como você sabe, é um número bem grande..."

"Eu preciso saber quem está pagando o aluguel. Eu realmente não me importo com quem está morando, apenas quem está pagando o aluguel. Nós os chamamos de responsável pelo pagamento. Somente um responsável por cada unidade. Eu preciso de seus nomes e números de seguridade social. Um responsável pode pagar o aluguel de mais de uma unidade? Não vejo por que não..."

"Preciso de uma porção de dados de cada unidade. Número de quartos, por exemplo. Se tem ou não um fogão e lavadora de pratos. E quais vagas de estacionamento estão atribuídas a cada unidade. Número? Claro, as vagas são numeradas, mas elas não combinam com os números das unidades, e não podemos fazer nada para mudar isso. Unidade 103, por exemplo, tem as vagas 1201 e 1202. E não podemos atribuir uma vaga de estacionamento para mais de uma unidade."

"Algumas dessas unidades nós chamamos unidade de luxo, tipo L. Outras são tipo N, de normal, e existe uma porção de unidades econômicas, tipo E. O valor do aluguel depende do tipo de unidade."

"Sim, características. Características adicionais. Temos tudo codificado mas sempre estamos pensando em novas siglas. Como o quê? Como VP para 'vista da piscina', e PT para 'perto das quadras de tênis'. Nós queremos manter um registro deste tipo de coisa. Quero listar todas estas características e eu quero também encontrar quais unidades tem tais características e quais não tem..."

"Por que você troca tanto de canetas? O que é que você está desenhando aí? O professor de Avaliação do ERA que você me fez participar disse que deveria usar preto antes de usar vermelho, e que deveria explicar como as tabelas são. Na realidade, vou definir os dados de amostragem..."

4.2. Entrevista 2

Leia a seguinte situação e desenhe o diagrama de entidade-relacionamento resultante da entrevista com o Gerente da Loja de Eletrônicos.

O gerente da loja diz:

"Temos uma grande confusão aqui. Não sabemos a quem pertence este CD player e também não sabemos a quem pertence este forno. De certa forma, teremos que ter um sistema melhor de rastreamento que identifique o que o cliente trouxe e o serviço a ser executado. O nome e o telefone do cliente nós gravamos..."

"Precisamos de um sistema melhor de rastreamento de técnicos também. Seus nomes e em quais atividades eles estão trabalhando em determinado momento, e quanto tempo eles levam para o cálculo da cobrança de horas. Claro que um técnico pode trabalhar numa atividade, e esta atividade pode demandar mais de um técnico para ser executada. Não, eu realmente não me importo com quem tem uma atividade atribuída mas eu gostaria de saber se um serviço já foi executado ou não, a atividade quero dizer. Apenas um código para indicar; SIM significando que a atividade já foi executada e NÃO para uma atividade ainda não executada."

"Como diferenciar um equipamento do outro? O sistema irá numerá-los e vamos

gravar o número em algum lugar que o cliente não possa ver. Acho que estaria bem TV para televisão, RD para rádio, CP para computador. O que é o que? Não, cada equipamento é de um único tipo. Nós precisamos de uma descrição para cada equipamento também. Esses códigos não são suficientes...”

“Temos de ter um acompanhamento do teste do equipamento também. Estes multitestes superTech custam por volta de \$2000 por peça. Temos 11 deles agora. São numerados de 101 até 111. O que eu quero saber é qual técnico está usando qual multitteste. E em qual bancada ele está trabalhando. O que você disse? Certo, somente um equipamento por técnico. E somente um técnico pode utilizá-lo por vez...”

“As bancadas? Bem, como pode ver, são grandes. Colocamos números no final de cada mesa. Temos quatro ou cinco técnicos trabalhando numa bancada. Cada técnico é sempre designado para uma única bancada...”

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.

Módulo 9

Banco de Dados



Lição 3

Projeto Lógico do Banco de Dados

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

1. Objetivos

Nesta lição discutiremos o projeto lógico do banco de dados. Embora existam muitos modelos de bancos dados que podem ser utilizados, esta lição focará no modelo de banco de dados relacional por dois motivos:

1. O modelo de dados relacional é amplamente utilizado nas aplicações de banco de dados
2. Certos princípios que se aplicam a outros modelos também se aplicam ao modelo relacional

Para entender o modelo de banco de dados relacional, certos conceitos matemáticos serão revistos. Além disso, o processo de transformar o modelo conceitual em modelo relacional será apresentado. Mais importante ainda, o conceito de normalização, integridade e restrições serão lecionadas.

Ao final desta lição, o estudante será capaz de:

- Através da lógica de dados, verificar como projeto do modelo conceitual (geralmente sob a forma de um Modelo Entidade-Relacionamento) é transformado em um projeto de banco de dados
- Compreender o modelo de dados relacionais, a álgebra relacional e o cálculo relacional
- Definir uma boa estrutura das relações através do conceito de normalização
- Conhecer as etapas na transformação de um MER em um conjunto de relações estruturadas

2. Projeto Lógico do Banco de Dados

É o processo de transformação do modelo de dados conceitual em um modelo lógico de dados que pode ser implementado em um sistema de gerenciamento de banco de dados. O resultado é um **modelo lógico de dados**, que é um projeto que está de acordo com os dados do modelo para uma classe de sistema de gerenciamento de banco de dados. Até agora existem quatro tipos de bases de dados lógicas. São eles:

1. **Modelo Hierárquico.** Registros estão organizados em uma estrutura que lembra uma árvore de cabeça para baixo. Os termos pai e filho são muitas vezes utilizados na descrição do modelo. Uma propriedade importante é que um registro filho pode estar relacionado a um único progenitor. A figura mostra um exemplo.

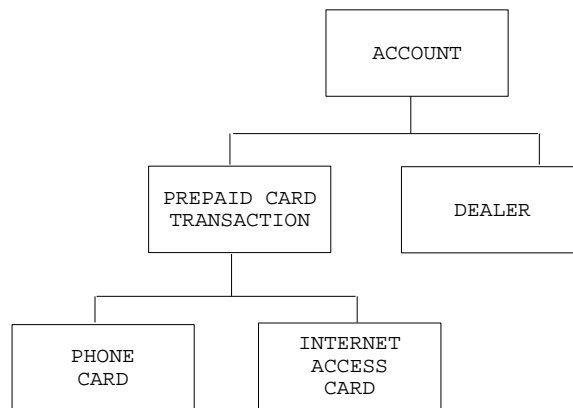


Figura 1: Modelo Hierárquico

Neste exemplo, ACCOUNT é o pai dos registros PREPAID CARD TRANSACTION e DEALER, enquanto PHONE CARD e INTERNET ACCESS CARD têm PREPAID CARD TRANSACTION como seu registro pai.

2. **Modelo de Rede.** É semelhante ao modelo hierárquico exceto que não há nenhuma distinção entre registros pai e filho. Qualquer tipo de registro pode ser associado a um número arbitrário de diferentes tipos de registro. Foi desenvolvido principalmente para superar a limitação no âmbito de aplicação do modelo hierárquico. A figura a seguir mostra o modelo da rede para a transação com cartão.

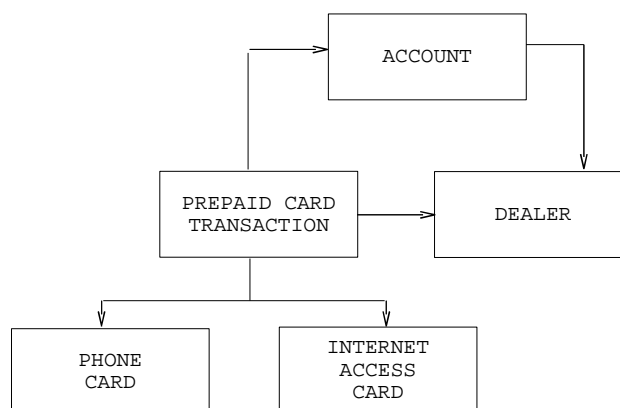


Figure 2: Modelo da Rede

Neste exemplo, PREPAID CARD TRANSACTION tem uma conta pertencente a um DEALER. Um PREPAID CARD TRANSACTION envolve tanto um PHONE CARD ou INTERNET ACCESS CARD.

3. **Modelo Relacional.** Os dados são representados sob a forma de entidades com linhas e colunas. Não há dados físicos representativos das associações entre as entidades, em vez disso as associações são representadas por valores lógicos que estão armazenados no interior nas colunas.

ACCOUNT

<u>CellPhoneNo</u>	PIN	Balance	Limit	Status	Type
09192345678	1234	\$500,00	\$2.500,00	ACT	2
09174561234	2345	\$100,00	\$2.000,00	ACT	2
09205467234	4523	\$25.000,00	\$300.000,00	ACT	1
09165647342	7812	\$30.000,00	\$300.000,00	ACT	1

STATUS

<u>CODE</u>	DESCRIPTION
ACT	Active Account
BAR	Barred Account
TER	Terminated Account

TYPE

<u>CODE</u>	DESCRIPTION
1	Dealer Account
2	Direct Reseller

PHONECARDTRANSACTION

<u>CELLPHONENO</u>	<u>CARDNO</u>	LOADDATE	RECIPIENT
09205467234	2346253782	DEC-24-2006	9223456173
09205467234	8736237634	DEC-24-2006	9178746345

Figura 3: Modelo Relacional

A Figura 3 representa a transação *Prepaid Card* como um modelo relacional. Neste exemplo, a entidade ACCOUNT possui chaves estrangeiras STATUS e TYPE cujos valores são chaves primárias de STATUS (coluna CODE) e TYPE (coluna CODE). Isto significa que STATUS e TYPE não podem ter valores que não estejam presente em STATUS e TYPE nas entidades respectivas.

4. **Modelo Orientado a Objeto.** Atributos de dados e métodos que operam esses atributos são encapsulados em estruturas chamadas objetos. Figura 4 mostra o modelo de objeto para a transação *Prepaid Card*. Seis objetos são definidos: DEALER, ACCOUNT, PREPAID CARD TRANSACTION, PREPAID CARD, PHONE CARD e INTERNET ACCESS CARD.

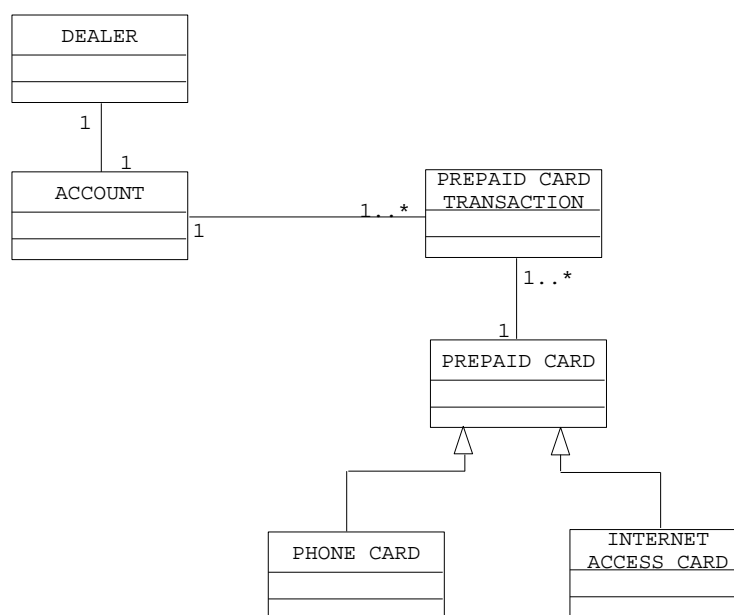


Figura 4: Modelo de Classes

3. Modelo de Dados Relacionais

O modelo de dados relacional foi introduzido pela primeira vez por E.F. Codd, um matemático especializado, em 1970. Baseou-se no conceito matemático de relação, que é fisicamente representado como uma entidade e utiliza terminologias da matemática, especificamente, ao definir a teoria e predicados lógicos. Nesta seção, terminologias e conceitos estruturais do modelo relacional serão discutidos.

Existem três componentes do modelo relacional. São eles:

1. A estrutura de dados é uma coleção de objetos ou relações que armazenam dados. Os dados são organizados sob a forma de entidades ou relações
2. A manipulação de dados consiste em operações (tais como os incorporados na linguagem SQL) que são usados para manipular os dados armazenados nas relações. Elas são baseadas no conjunto dos operadores que atuam sobre as relações
3. A integridade dos dados inclui facilidades para especificar as normas que mantêm a integridade dos dados. É utilizado para precisão e a coerência

Uma base de dados é uma coleção de relações ou entidades. As relações no interior da base de dados deverão ser devidamente estruturadas. A adequação é conhecida como normalização, que será discutida mais tarde, neste capítulo.

A relação é uma tabela de dados bidimensional. É constituída por um conjunto de linhas e colunas.

Uma linha ou tupla representa todos os dados necessários para uma determinada relação. Corresponde a um único registro. Na relação representa um grupo horizontal de células.

Uma coluna corresponde a um atributo de uma relação. A relação representa um grupo vertical de células. A Figura 5 mostra um exemplo de uma relação. Cada registro, ou linha, é identificado por uma chave primária, que é uma coluna ou atributo que identifica exclusivamente registros, ou seja, o valor desta coluna deve ser único e não nulo na relação. A chave composta é a chave primária que é constituída por mais de um atributo. Uma chave estrangeira é um atributo de uma relação de um banco de dados que serve como uma chave primária de uma outra função na mesma base de dados.

ACCOUNT

<u>CellPhoneNo</u>	PIN	Balance	Limit	<u>Status</u>	<u>Type</u>
09192345678	1234	\$500,00	\$2.500,00	ACT	2
09174561234	2345	\$100,00	\$2.000,00	ACT	2
09205467234	4523	\$25.000,00	\$300.000,00	ACT	1
09165647342	7812	\$30.000,00	\$300.000,00	ACT	1

Figura 5: Relação de Contas

Neste exemplo, o nome da relação é ACCOUNT. As colunas de ACCOUNT são CellPhoneNo, PIN, Balance, Limit, Status e Type. Um exemplo de uma linha ou tupla é constituído pelos seguintes valores: (09192345678, 1234, \$500.00, \$2.500,00, ACT, 2). A chave primária da entidade é CellPhoneNo, e está sublinhada com um traço sólido. As chaves estrangeiras da relação são Status e Type, e estão sublinhadas com um traço pontilhado.

A relação, ou tabela, pode ser expressa pela seguinte abreviação, ou notação:

ACCOUNT(CellPhoneNo, PIN, Balance, Limit, Status, Type)

Um domínio é o conjunto de valores admissíveis para um ou mais atributos. São extremamente poderosos como característica do modelo relacional uma vez que cada atributo em um banco de dados relacional é definido em um domínio. Permitem aos usuários definir em um local central o significado das colunas e a fonte de valores que pode conter e, assim, fornecer mais informação para o sistema na execução de operações relacionais, e as operações que são semanticamente incorretas podem ser evitadas. Como exemplo, não faz sentido comparar o número de telefone

móvel com o último nome do proprietário da conta, apesar de terem o mesmo domínio, ou seja, valores. A Figura 6 mostra um exemplo para o domínio da relação ACCOUNT.

Coluna	Domínio	Significado	Definição
CellPhoneNo	Número do Telefone	Conjunto de todos os números móveis válidos nas Filipinas	Número: tamanho 11
PIN	Número de Identificação Pessoal	Conjunto de todos os quatro dígitos do número de identificação	Número: tamanho 4
Balance	Saldo remanescente	Conjunto que representa valor de saldo de conta que deverá ser positivo	Dinheiro: tamanho 8, Faixa: 0 a \$500.000,00
Limit	Limite de Conta	Conjunto que representam o limite da conta em valor monetário	Dinheiro: tamanho 8, Faixa: 0 a \$500.000,00
Status	Status atual Conta	Conjunto de três caracteres que representa o status da conta	Caractere: tamanho 3
Type	Tipo de Conta	Conjunto de um dígito do número que representa o tipo de conta (concessionário ou revendedor direto)	Número: tamanho 1, Faixa: 1 a 2

Figura 6: Domínio de Relação de Contas

A estrutura da relação, juntamente com a especificação dos domínios, é chamada intenção de uma relação. **É fixa**, a menos que o significado da relação mude, como a adição de novas colunas. **A extensão da relação** são as informações de uma relação que mudam ao longo do tempo, tais como as informações sobre as linhas da relação.

O grau de uma relação é o número de colunas que contém. A relação ACCOUNT tem seis atributos. Isso significa que cada linha da entidade tem seis tuplas, ou seja, seis colunas. A relação com apenas um atributo é conhecida como uma relação unária. A relação com dois atributos é conhecida como uma relação binária. A relação contendo três atributos é conhecida como uma relação ternária. Mais de três atributos se chama n-ária relação. O grau de relação faz parte da intenção da relação.

A cardinalidade de uma relação é o número de tuplas que esta contém. Muda a medida que linhas são adicionadas ou retiradas da tabela. Trata-se de uma propriedade da extensão da relação e é determinada a partir de um exemplo em particular a qualquer momento.

Há terminologias alternativas. A Figura 7 mostra isso. A segunda alternativa é a mais utilizada nos aspectos físicos do sistema de gerenciamento de banco de dados relacional.

Termos	Alternativa 1	Alternativa 2
Relação	Tabela	Entidade
Tupla	Linha	Registro
Atributo	Coluna	Campo

Figura 7: Terminologia Alternativa

3.1. Relações Matemáticas

Para compreender o verdadeiro significado da expressão “relação”, uma revisão de alguns conceitos matemáticos é necessária. Suponha que tenhamos dois conjuntos, D1 e D2, onde

$$D_1 = \{2, 4\} \text{ e } D_2 = \{1, 3, 5\}$$

O produto cartesiano desses dois conjuntos, escrito como $D_1 \times D_2$, é o conjunto de todos os pares ordenados tais que o primeiro elemento é um membro da D1 e o segundo elemento é um

membro da D2. É representada como se segue:

$$D_1 \times D_2 = \{(2,1), (2,3), (2,5), (4,1), (4,3), (4,5)\}$$

Qualquer subconjunto do produto cartesiano é uma relação. Por exemplo, podemos produzir o seguinte:

$$R = \{(2,3), (4,1)\}$$

Pode-se especificar os pares ordenados que farão parte da relação, dando uma condição para a sua seleção. Exemplo, a relação R deve conter um par ordenado quando o segundo elemento é igual a três (3). Isto pode ser especificado como segue:

$$R = \{(x,y) \mid x \in D_1, y \in D_2, \text{ e } y = 3\}$$

Pode-se facilmente ampliar a noção de uma relação a três conjuntos. Considerando D1, D2 e D3 como três conjuntos. O produto cartesiano é D1 X D2 X D3 quando se trata de um conjunto de todos os triplos ordenados tal que o primeiro elemento é a partir de D1, o segundo elemento é de D2 e do terceiro elemento é de D3. Qualquer subconjunto do produto cartesiano destes é uma relação. Por exemplo, suponha que temos:

$$\begin{aligned} D_1 &= \{1, 3\} \\ D_2 &= \{2, 4\} \\ D_3 &= \{5, 6\} \\ D_1 \times D_2 \times D_3 &= \{(1,2,5), (3,2,5), (1,4,5), (3,4,5), \\ &\quad (1,2,6), (3,2,6), (1,4,6), (3,4,6)\} \end{aligned}$$

Em geral, seja D_1, D_2, \dots, D_n n um conjunto de domínios. Seu produto cartesiano é definido como:

$$D_1 \times D_2 \times \dots \times D_n = \{(d_1, d_2, \dots, d_n) \mid d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n\}$$

normalmente é escrito como:

$$\prod_{i=1}^n D_i$$

Qualquer conjunto de n-tuplas deste produto cartesiano é uma relação entre n conjuntos. Na definição dessas relações, deve-se especificar os conjuntos, ou domínios, a partir do qual se escolhe um valor.

3.2. Relações de Bancos de Dados

Aplicando estes conceitos para o banco de dados, um esquema de relação é um nome de relação seguido por uma série de colunas e os pares do nome do domínio. Seja A_1, A_2, \dots, A_n colunas nos domínios D_1, D_2, \dots, D_n . Portanto, o conjunto $\{A_1:D_1, A_2:D_2, \dots, A_n:D_n\}$ é um esquema de relação.

Uma relação R definida por um esquema de relação S é um conjunto de mapeamentos dos nomes das colunas correspondentes aos seus domínios. Assim, a relação R é um conjunto de n-tuplas.

$$(A_1:D_1, A_2:D_2, \dots, A_n:D_n) \text{ tal que } d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$$

Cada elemento do n-tuplas consiste de uma coluna e um valor para esse atributo. Com relação a ACCOUNT, como exemplo, o esquema é o seguinte:

(**CellPhoneNo**:09192345678, **PIN**:1234, **Balance**:\$500,00, **Limit**:\$2.500,00, **Status**:ACT, **Type**:2)

No modelo relacional, as relações têm propriedades que deve existir. Ou seja, são os seguintes:

1. A relação tem um nome que é único na relação de todos os outros nomes dentro do banco de dados
2. Cada campo de uma relação contém exatamente um valor atômico (único)

3. Cada coluna tem um nome diferente
4. O valor de uma coluna são todos de um mesmo domínio
5. A ordem das colunas não tem qualquer significado
6. Cada linha é única, ou seja, não existem registros duplicados
7. A ordem da fila não tem qualquer significado, teoricamente. Na prática, a ordem das linhas tem efeito sobre a eficiência de acesso aos registros.

Quando bem construídas as relações contêm mínima redundância e permitem inserções, alterações e exclusões de linhas sem quaisquer erros ou inconsistências. Três tipos de anomalias devem ser evitadas quando se constrói relações de um banco de dados. São elas:

1. Anomalias de Inserção
2. Anomalias de Exclusão
3. Anomalias de Modificação

Para ilustrar anomalias, considere a seguinte figura em relação ao que não foi bem construído:

ACCOUNT1

CELLPHONENO	Type	CARDNO	LOADDATE	RECIPIENT
09205467234	2	2346253782	DEC-24-2006	9223456173
09205467234	2	8736237634	DEC-24-2006	9178746345
09165647342	1	9347536455	DEC-25-2006	9165234234
09174561234	1	2345234123	DEC-25-2006	9226352345
09174561234	1	6354239564	DEC-27-2006	9185343838

Figura 8: Tabela de Anomalias

1. **Anomalia de Inserção.** Considere a adição de um novo número de telefone celular. Nesta entidade, a chave primária é CELLPHONENO e CARDNO. O registro não pode ser adicionado porque CARDNO porque não foi informado ainda. Chaves primárias não podem ser nulas ou inexistentes.
2. **Anomalia de Exclusão.** Considere a possibilidade de excluir o registro com o valor de CELLPHONENO 09165647342. Uma vez removida, o CARDNO 9347536455 será perdido.
3. **Anomalia de Modificação.** Considere a modificação do tipo de conta com a mudança em CELLPHONENO de 09205467234 para 1. Neste exemplo, dois registros precisam ser atualizados.

3.3. Chaves Relacionais

Para identificar uma linha dentro de um relação, primeiro devemos identificar valores sem atributos iguais dentro desta relação. Uma **superchave** é uma coluna ou um conjunto de colunas que identifica exclusivamente uma linha dentro de uma relação.

Uma chave candidata é uma **superchave** tal que nenhum subconjunto é uma **superchave** dentro da relação. Uma chave candidata K, para uma relação R, possui duas propriedades:

1. **Única ou Distinta.** Em cada linha de R, os valores de K identificam-na exclusivamente
2. **Irreduzível.** Nenhum subconjunto de K tem uma propriedade de singularidade

Uma chave composta é uma chave candidata que consiste em duas ou mais colunas.

Uma **chave primária** é uma chave candidata que é escolhida para identificar exclusivamente uma linha dentro de uma relação. Considerando que uma relação não possui nenhuma fila duplicada, sempre é possível identificar cada fila que usa a chave primária exclusivamente. Isto significa que uma relação sempre possui uma chave primária. No pior caso seria uma relação que contém todas as colunas como chave primária; porém, normalmente em algum subconjunto

menor de colunas é suficiente para distinguir as filas. As chaves candidatas que não são selecionadas como chaves primárias são conhecidas como chaves substitutas. Considere a relação de PHONECARDTRANSACTION. A chave primária também é uma chave composta que consiste nas colunas CELLPHONENO e CARDNO.

PHONECARDTRANSACTION			
<u>CELLPHONENO</u>	<u>CARDNO</u>	LOADDATE	RECIPIENT
09205467234	2346253782	DEC-24-2006	9223456173
09205467234	8736237634	DEC-24-2006	9178746345

Figura 9: Relação PHONECARDTRANSACTION

Uma **chave estrangeira** é uma coluna ou um conjunto de colunas dentro de uma relação que emparelha a chave candidata de alguns (possivelmente no mesma) relação. Quando uma coluna aparecer em mais de uma relação, normalmente isto representa uma relação entre as filas das duas relações.

Por exemplo, considere as relações ACCOUNT e PHONECARDTRANSACTION. A coluna de CELLPHONENO aparece em ambas relações. Isto indica um relacionamento que uma conta pode ter transações de cartão telefônico.

4. Conceito de Normalização

Para construir relações bem-estruturadas, o conceito de normalização é usado. Normalização é o processo de converter estruturas complexas de dados em estruturas estáveis e simples. É realizado em fases chamadas de **formas normais**. Está baseado na análise das dependências funcionais.

Uma **forma normal** é um estado de uma relação que pode ser determinado aplicando-se regras simples relativas às dependências daquela relação.

Dependências funcionais são relações particulares entre dois atributos. Para qualquer relação **R**, o atributo **B** é funcionalmente dependente do atributo **A**, se, para todo exemplo válido de **A**, aquele valor de **A** determina exclusivamente o valor de **B**. Pode ser representado por:

$$A \rightarrow B$$

Onde **A** é um determinante.

O determinante **A** é o atributo achado ao lado esquerdo da seta em uma dependência funcional. Pode ser usado como a chave primária da relação. Como um exemplo, considere a relação:

EMPLOYEE_COURSE (EMPLOYEE_ID, COURSE, DATE_COMPLETED)

Neste exemplo, EMPLOYEE_ID e COURSE formam a chave composta que determina DATE_COMPLETED funcionalmente. A dependência funcional é escrita a seguir:

EMPLOYEE_ID, COURSE \rightarrow DATE_COMPLETED

DATE_COMPLETED de um COURSE é determinada pelo EMPLOYEE_ID que o efetuou.

Dependência funcional possui as seguintes regras:

1. **Regra Reflexiva.** Um atributo é funcionalmente dependente de si mesmo. É representado como:

$$X \rightarrow X$$

2. **Regra de Aumento.** Se $X \rightarrow Y$, então $X, Z \rightarrow Y$. Considere a dependência funcional:

STUDENT NUMBER \rightarrow STUDENT NAME

Pela regra de aumento, um estudante pode ter a seguinte dependência funcional:

STUDENT NUMBER, COURSE \rightarrow STUDENT NAME

3. **Regra de União.** Se $X \rightarrow Y$ e $X \rightarrow Z$, então $X \rightarrow Y, Z$. Considere as duas dependências funcionais:

STUDENT NUMBER \rightarrow STUDENT NAME

STUDENT NUMBER \rightarrow ADDRESS

Pode-se combinar em uma única dependência funcional, como a seguir:

STUDENT NUMBER \rightarrow STUDENT NAME, ADDRESS

4. **Regra de Decomposição.** Se $X \rightarrow Y$ e Z é um subconjunto de Y, então $X \rightarrow Z$
5. **Regra de Transitividade.** Se $X \rightarrow Y$ e $Y \rightarrow Z$, então $X \rightarrow Z$. Considere as duas dependências funcionais:

STUDENT NUMBER \rightarrow MAJOR

MAJOR \rightarrow ADVISOR

Da regra de transitividade, podemos ter a seguinte dependência funcional:

STUDENT NUMBER → ADVISOR

6. **Regra de Pseudotransitividade.** Se $X \rightarrow Y$ e $Y, Z \rightarrow W$, então $X, Z \rightarrow W$. Considere as duas dependências funcionais seguintes:

STUDENT NUMBER → MAJOR

MAJOR, CLASS → ADVISOR

Da regra de pseudotransitividade, pode-se ter a seguinte dependência funcional:

STUDENT NUMBER, CLASS → ADVISOR

4.1. Passos em Normalização

Normalização pode ser entendida e executada em estágios. Cada estágio corresponde a uma forma normal que é um estado de uma relação que resulta da aplicação de simples regras em relação a dependências funcionais. Como representação de nossa estrutura de dados, usaremos o diagrama de *Warnier-Orr* para mostrar as dependências funcionais das colunas. O diagrama de Warnier-Orr usa linhas para separar o que está sendo descrito e os itens que o compõem. A Figura 10 mostra os diagramas e o significado dos diagramas.

Diagrama	Significado
D — [A B C]	D consiste de A, B, e C
E — [F (or) G (or) H]	Seleciona somente um dos itens Que pode ser F, G, ou H
E — [F (and/or) G (and/or) H]	Qualquer um dos itens Que pode ser qualquer combinação.
J (1:3) — [K L M O (opcional)]	Iteração ou Repetição J consiste de 1 a 3 iterações de K, L, M ou O que é opcional

Figura 10: Diagrama Warnier-Orr

Considere o seguinte visão da Figura 11 que possui seu correspondente com o diagrama *Warnier-Orr* na Figura 12. Normalização será usada para definir um conjunto bem definido de relações.

CONTRACT SCREEN				
Contract No.: 49836		Customer Name: Tipp, Q.		
Customer No.: TIP32		Address: 214 th Avenue Cubao Quezon City		
Contract Date: 16 Feb 1993				
Sales Office: CUB001				
Appliance No.	Model No.	Manufacturer	Date	Services
140446	CTV27	Hitachi	26 Aug 1992	2
310543	VHS03	JVC Phil	03 Feb 1993	0
140221	CTV35	Sony	01 Mar 1993	1

Figura 11: Lista de Contrato por Cliente

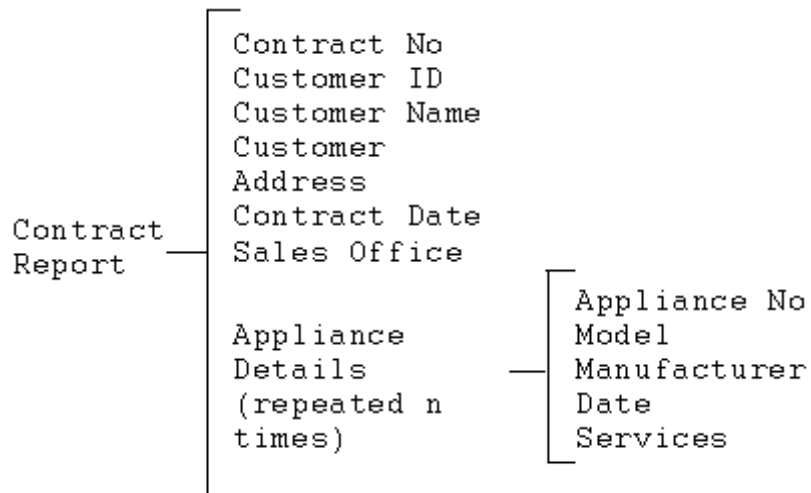


Figura 12: Diagrama Warnier-Orr da Lista de Contrato por Cliente

4.1.1. Primeira Forma Normal (FNF)

Uma relação está na primeira forma normal (FNF) se não contém grupos repetidos. O que segue é o procedimento:

```

** Un-normalized Form to First Normal Form
** Objective: Remove Repeating Groups
From non-repeated data items
  Identify a Primary Key
  IF none is found
    Create a Primary Key
  ENDIF
IF there is/are repeating group (s)
  DO WHILE there are repeating groups
    From the outer-most repeating groups,
      Identify a Group Key
      IF none is found
        Create a Group Key
      ENDIF
  ** Break un-normalized form into two relations
  Relation 1 = Primary Key + non-repeating data items
  Relation 2 = Primary Key + Group Key + group data items
  IF Relation 2 still has repeating groups
    Primary Key = Old Primary Key + Group Key
  ENDIF
ENDDO
ENDIF
Rename Resultant Relation
Draw First Normal Form Data Model
** Relations are now in the First Normal Form

```

Diretivas para escolher uma chave:

- Deve identificar unicamente componentes, itens ou entidades representadas em dados não normalizados
- Não deve estar repetida dentro de dados não normalizados
- A chave deve ser única para cada ocorrência de um grupo de dados não normalizados
- Se existe uma escolha a ser feita entre itens de chave, escolha baseado em:
 - um campo, ao invés de vários
 - numérico, ao invés de alfanumérico
 - comprimento fixo, ao invés de comprimento variável

- formato fixo, ao invés de formato variável
- itens curtos, ao invés de itens longos

O campo APPLIANCE DETAIL é considerado um grupo repetido. Foi removido para formar outra relação com sua chave primária própria. A Figura 13 mostra as relações resultantes.

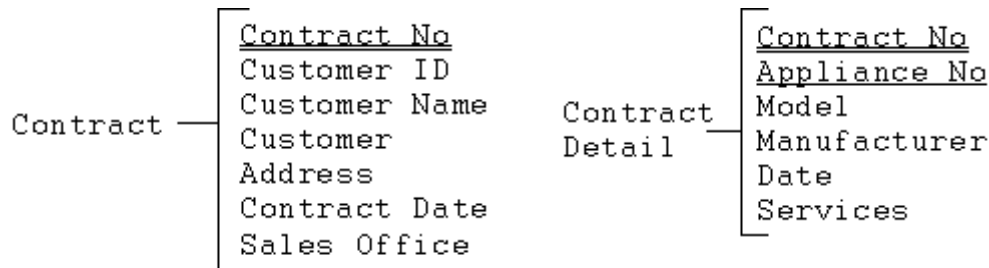


Figura 13: Primeira Forma Normal (FNF) – Lista de Contratos por Cliente

4.1.2. Segunda Forma Normal (SNF)

Uma relação está na Segunda Forma Normal se está em FNF e cada atributo não-chave é completamente funcionalmente dependente da chave primária. Um **atributo não-chave** é um atributo que não é a chave primária ou é parte de uma chave composta.

As relações identificadas não podem ser usadas como base para o projeto de dados, pois possuem propriedades indesejáveis que levam a atualizações anômalas. Para mudar de FNF para SNF, remova as dependências de partes da chave. Envolve examinar as relações que têm uma chave composta ou concatenada e para cada campo faça as seguintes perguntas: Pode o campo ser unicamente identificado por uma parte da chave, ou é necessária a sua totalidade (chave composta) ?

```

** First Normal Form to Second Normal Form Procedure
** Objective: Remove attributes not dependent on the Relation's Composite
** keys.
From the First Normal Form Relations
Identify Relations with Composite Keys
FOR EACH Relation with Composite Keys
  Identify Composite Key
  Mark with a left-side double bracket
  Identify Part Keys
  Mark with a right-side single bracket
FOR EACH Non-key data item
  IF data item is dependent on the Composite Key
    Draw a left-side line to Composite Key
  ELSE
    Draw a right-side line to Part Key
  ENDIF
ENDFOR
** Break First Normal Form Relation into two Relations
Relation 1 = Composite Key + Left-side data items
Relation 2 = Part Key + Right-side data items
ENDFOR
Rename Resultant Relations
Draw Second Normal Form Data Model
** Relations are now in the Second Normal Form
  
```

A relação CONTRACT tem somente uma chave, conseqüentemente, já está na segunda forma normal. Entretanto, CONTRACT DETAIL possui um chave composta, de modo que ela deve estar sujeita ao procedimento SNF.

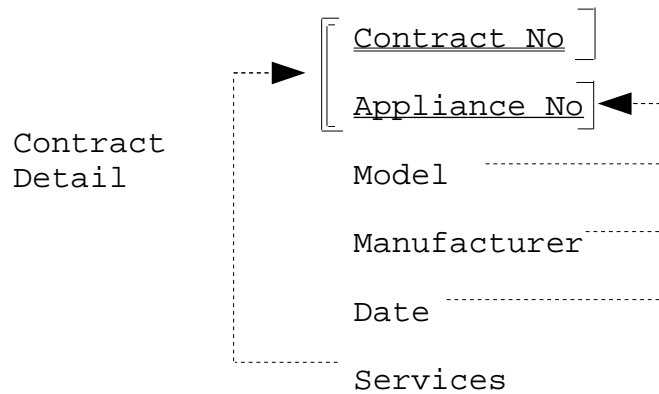


Figura 14: Dependências Funcionais de Contract Details

Somente SERVICES é completamente funcionalmente dependente da chave composta. O resto (MODEL, MANUFACTURER e DATE) é funcionalmente dependente de APPLIANCE NO. As relações resultantes podem ser vistas na Figura 15.

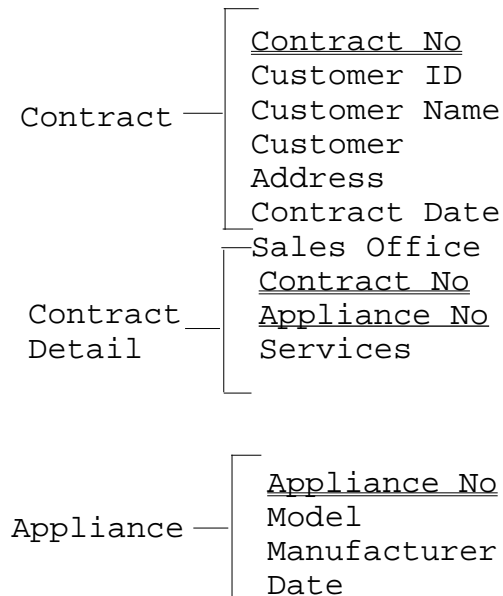


Figura 15: Segunda Forma Normal da Lista de Contratos por Cliente

4.1.3. Terceira Forma Normal (TNF)

Uma relação está na terceira forma normal se é SNF e não existem dependências transitivas. Uma **dependência transitiva** em uma relação é uma dependência funcional entre dois (ou mais) atributos não-chave.

Algumas anomalias podem ainda existir nos dados. O terceiro passo de normalização lida com a identificação de chaves estrangeiras e atributos, removendo-os de modo que não haverá mais dependências entre os itens de dados ou chaves.

```

** Second Normal Form to Third Normal Form Procedure
** Objective: Remove Foreign Keys and their attributes
FOR EACH Second Normal Form Relation
  IF there is a possible Foreign Key
    Identify Foreign Key
    IF Foreign Key can be replaced with a Code
      Add Code to data items
      Replace Foreign Key with Code
    ENDIF
  
```

```

FOR EACH Non-Key data item
  IF data item is dependent on the Primary/Composite Key
    Draw a left-side line to Primary/Composite Key
  ELSE
    Draw a right-side line to Foreign Key
  ENDIF
ENDFOR
** Break Second Normal Form into two relations
Relation 1 = Primary/Composite Key + Foreign Key + left-side data items
Relation 2 = Foreign Key + right-side data items
ENDIF
Rename resultant relation
ENDFOR
Draw Third Normal Form Data Model
** Relations are now in the Third Normal Form

```

Olhando as relações SNF, as possíveis chaves estrangeiras são CONTRACT e APPLIANCE. Para CONTRACT, CUSTOMER ID é considerada uma chave estrangeira. Para APPLIANCE, MODEL é considerada uma chave estrangeira. A Figura 16 mostra a dependência transitiva das relações.

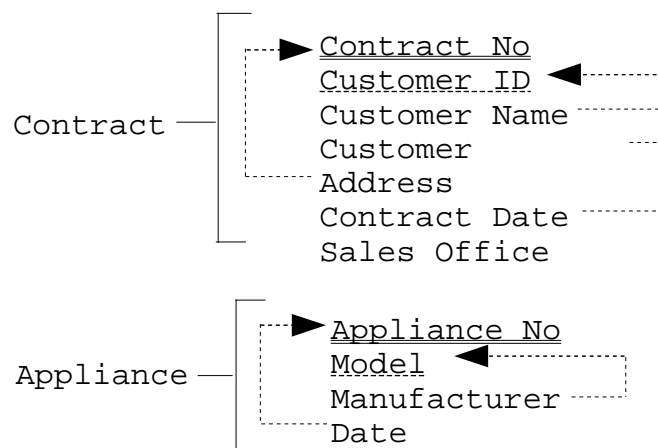


Figura 16: Dependência Transitiva da Lista de Contratos do Cliente

As relações resultantes são mostradas na Figura 17.

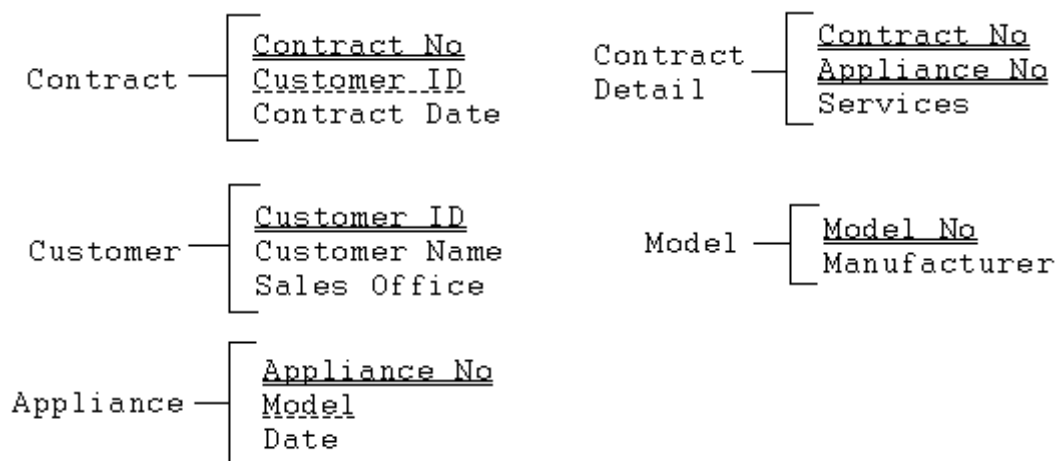


Figura 17: Terceira Forma Normal da Lista de Contratos por Clientes

Relações em TNF são suficientes na maioria das aplicações práticas de bancos de dados. Entretanto, TNF não garante que todas as anomalias foram removidas. Existem outras formas normais que permitem a remoção de certas anomalias.

4.1.4. Forma Normal de Boyce-Codd (BCNF)

Uma relação está em BCNF se, e somente se, cada determinante é um candidato a chave. Como revisão, um determinante é um atributo ou grupo de atributos no qual algum outro atributo é completamente e funcionalmente dependente. Certas pistas nos permitem determinar se uma relação viola a BCNF. São elas:

- A relação contém dois (ou mais) candidatos a chaves compostas
- O candidato a chave composta sobrepõe ou compartilha no mínimo um atributo em comum

Como exemplo, considere a relação definida na Figura 18. PATIENT_APPOINTMENT.

PATIENT_APPOINTMENT

<u>Patient_No</u>	<u>Interview_Date</u>	Interview_Time	Staff_No	Room_No
1077	May 13, 2004	9:30	105	101
1056	May 13, 2004	10:30	105	101
1065	May 13, 2004	1:00	137	102
1056	July 1, 2004	9:30	105	102

Figura 18: Relação Patient Appointment

As dependências funcionais da relação são:

Patient_No, Interview_Date → Interview_Time, Staff_No, Room_No
 Staff_No, Interview_Date → Patient_No
 Staff_No, Interview_Date → Room_No

Os determinantes de duas dependências funcionais são candidatos a chave composta da relação. Entretanto, o último não é porque ROOM_NO é mais funcionalmente dependente em STAFF_NO, INTERVIEW_DATE do que PATIENT_NO, INTERVIEW_DATE. Para transformar a relação para BCNF, crie duas novas relações. A Figura 19 mostra as relações resultantes.

PATIENT_APPOINTMENT

<u>Patient_No</u>	<u>Interview_Date</u>	Interview_Time	Staff_No
1077	May 13, 2004	9:30	105
1056	May 13, 2004	10:30	105
1065	May 13, 2004	1:00	137
1056	July 1, 2004	9:30	105

ROOM_ASSIGNMENT

<u>Staff_No</u>	<u>Interview_Date</u>	Room_No
105	May 13, 2004	101
137	May 13, 2004	102
105	July 1, 2004	102

Figura 19: Relação BCNF Patient Appointment

Pode não ser sempre desejável transformar uma relação em BCNF. Se existe uma dependência funcional que não está preservada quando a decomposição é executada, não a converta em BCNF.

4.1.5. Quarta Forma Normal

Esta é a forma de normalização a qual é uma relação BCNF e contém dependências de multivalores não triviais. **Dependências de Multivalores (DMV)** representa uma dependência entre atributos A, B, e C numa relação que para cada valor de A, existe um conjunto de valores de B e C. No entanto, os conjuntos de valores de B e C são independentes. A notação de dependência de multivalores é a seguinte:

A ->> B
A ->> C

Uma DMV trivial é definida por DMV A ->> B em relação a R se satisfaz a seguinte condição:

- B é um subconjunto de A ou
- $A \cup B = R$

Uma DMV não-trivial acontece quando as regras especificadas numa DMV trivial são violadas. Por exemplo: B não é um subconjunto A ou $A \cup B$ não resulta em R.

Como exemplo, considere a relação `BRANCH_STAFF_CUSTOMER_ASSIGNMENT` na Figura 20. Considere as seguintes premissas.

1. Um `BRANCH` possui vários `STAFF`.
2. Um `BRANCH` possui vários `COSTUMERS`.
3. Não há nenhuma relação exclusiva entre `STAFF` and `CUSTOMER`. Por exemplo, Elaine Stan pode se inscrever tanto para Ann Bautista como para David Fort.

BRANCH_STAFF_CUSTOMER_ASSIGNMENT		
Branch_No	Staff_Name	Customer_Name
1003	Ann Bautista	Elaine Stan
1003	David Fort	Elaine Stan
1003	Ann Bautista	Mike Ross
1003	David Ford	Mike Ross

Figura 20: Relacionamento Branch-Staff-Customer

O DMV que existe nessa relação é:

Branch_No ->> Staff_Name
Branch_No ->> Customer_Name

Para normalizar, remova o DMV criando duas novas relações. Figura 21 Mostra o resultado.

BRANCH_STAFF	
<u>Branch_No</u>	Staff_Name
1003	Ann Bautista
1003	David Fort

BRANCH_CLIENT	
<u>Branch_No</u>	Customer_Name
1003	Elaine Stan
1003	Mike Ross

Figura 21: Quarta Forma de Normalização da relação Branch-Staff-Customer

4.1.6. Quinta Forma Normal

Uma relação que não há nenhuma dependência de união. **Dependência Lossless-Join** é uma propriedade de decomposição, a qual assegura que nenhuma linha falsa será gerada quando as relações forem reunidas numa junção natural. Considere a relação `SUPPLIER_SHIPMENT` na Figura 22. A relação descreve os itens providos pelo fornecedor (supplier) para uma loja (store) particular. Não suporta nenhuma restrição que certos tipos de fornecedores (suppliers) deveriam prover certos itens para uma loja (store) em particular. Tirando o fato que um determinado fornecedor pode prover todos itens requeridos para uma determinada loja. Por exemplo, para a loja 104, apenas o fornecedor 10 pode prover *sofa beds* para aquela loja. Embora o fornecedor 20 também possa prover *sofa bed*.

`SUPPLIER_SHIPMENT`

Store_No	Item_Description	Supplier_No
104	Sofa Bed	10
104	Computer Chair	20
116	Sofa Bed	20
116	Table	10
136	Computer Chair	30

Figura 22: Relacionamento Supplier-Shipment

Esta relação pode ainda adicionar linhas (104, Sofa Bed, 20) e (104, Computer Chair, 30). Para transformar a relação em 5NF, decompõe a relação em três que suportem as seguintes restrições:

- Items providos por um fornecedor (supplier).
- Lojas (Stores) que o fornecedor por suprir items.
- Items achados nas lojas.

STORE_ITEM

<u>Store_No</u>	<u>Item</u>
104	Sofa Bed
104	Computer Chair
116	Sofa Bed
116	Table
136	Computer Chair

SUPPLIER_ITEM

<u>Supplier_No</u>	<u>Item</u>
10	Sofa Bed
10	Computer Chair
20	Sofa Bed
20	Table
30	Computer Chair

STORE_SUPPLIER

<u>Store_No</u>	<u>Supplier_No</u>
104	10
104	20
116	10
116	20
136	30

Figura 23: Quinta Forma de Normalização- Store-Item-Supplier Shipment

A Figura 23 mostra o resultado dos relacionamentos.

5. Transformando E-R Diagramas em Relacionamentos

Como exemplo, iremos transformar o diagrama de Entidade-Relacionamento do capítulo anterior e será mostrado a seguir:

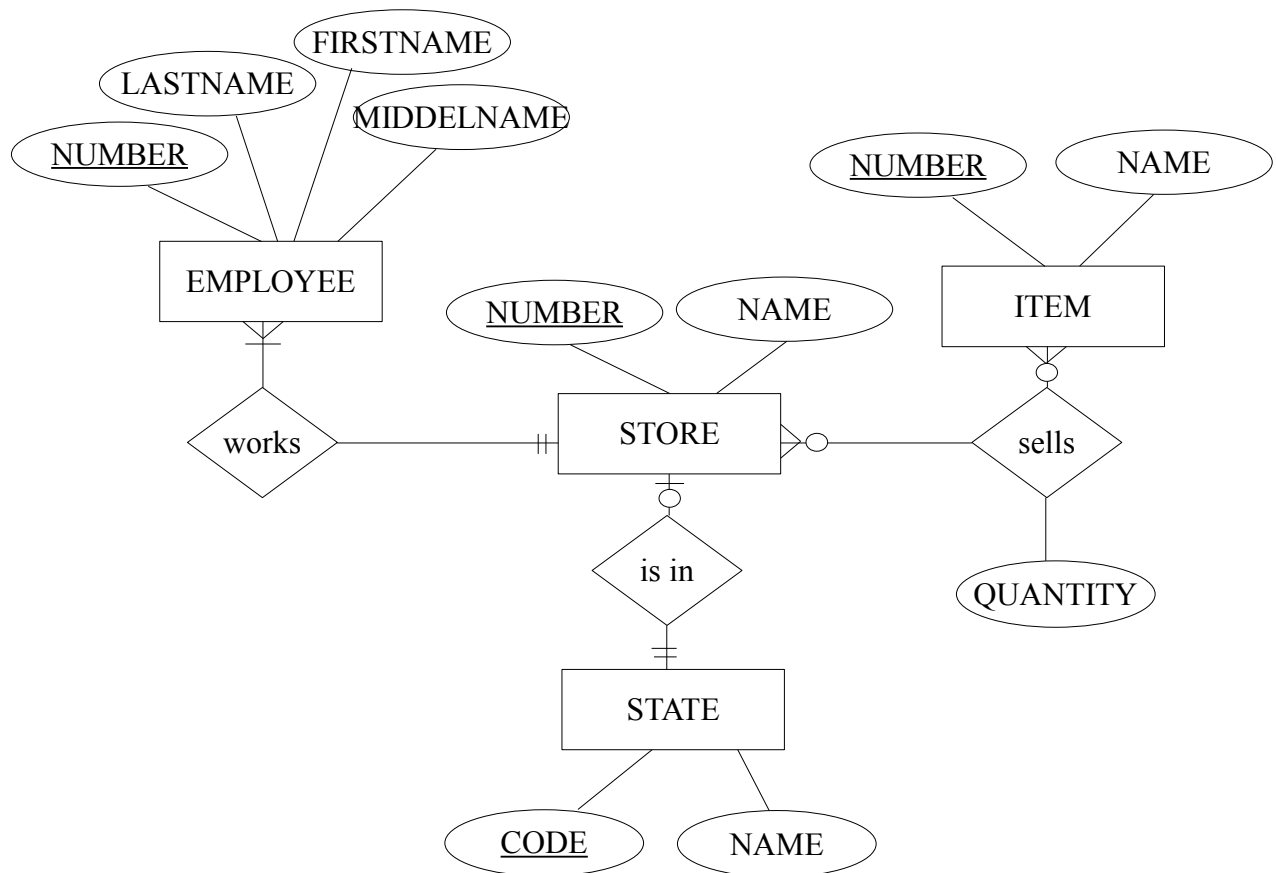


Figura 24: Diagrama Entidade - Relacionamento Organic Shop

5.1. PASSO 1: Representar Entidades

Cada tipo de entidade se torna uma relação. A chave primária da entidade se torna a chave primária da relação. Como revisão, chave primária deve ter as seguintes propriedades:

- O valor da chave deve ser um identificador único para cada linha da relação.
- A chave não pode ser redundante; ou seja, nenhum atributo da chave pode ser removido sem destruir identificação única.

Cada atributo não-chave da entidade se torna atributo não-chave da relação.

Como exemplo, considere a entidade modelada na Figura 25.

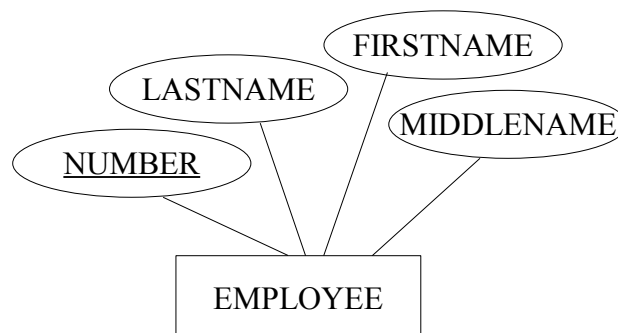


Figura 25: Employee DER

Podemos representá-la usando a tabala, shorthand or Warnier-Orr. No exemplo, a chave primária EMPLOYEE é Number enquanto os outros atributos (LastName, FirstName, MiddleName) são atributos não-chave da relação. O resultado da relação é mostrado na Figura 26.

EMPLOYEE			
<u>Number</u>	LastName	FirstName	MiddleName
5000	Stone	Benjamin	J.
5001	Cruz	Juan	M.
5002	Enriquez	Sheila	S.P.
5003	Choo	James	Y.
5004	Mendes	Lani	M.

EMPLOYEE(Number, LastName, FirstName, MiddleName)

Figura 26: Relação Employee

5.2. PASSO 2: Representando Relacionamentos

A representação dos relacionamentos depende do nível dos *relacionamentos* (unário, binário ou ternário) e as *cardinalidades dos relacionamentos* (mandatório um, opcionalmente zero etc.).

5.2.1. Representando um relacionamento Binário 1:N

No lado **N** da entidade, coloque a chave primária no lado **1** da entidade, tornando-a chave estrangeira. No exemplo da Figura 27, a chave primária de STORE (Number) se torna a chave estrangeira de EMPLOYEE (Store). Os valores dos atributos de Store contém valores achados no domínio de Number na relação de STORE.

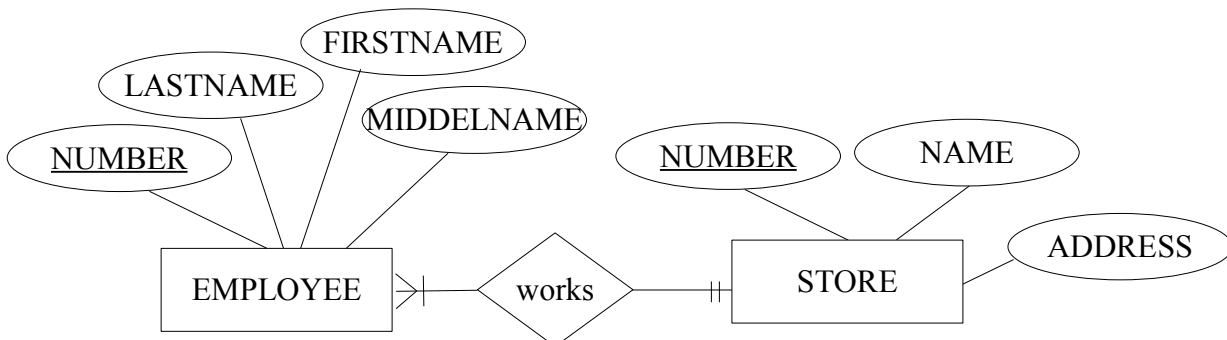


Figura 27: DER de Employee works in a store

O resultado da relação são mostrados na Figura 28.

EMPLOYEE				
<u>Number</u>	LastName	FirstName	MiddleName	<u>Store</u>
5000	Stone	Benjamin	J.	10
5001	Cruz	Juan	M.	10
5002	Enriquez	Sheila	S.P.	10
5003	Choo	James	Y.	10
5004	Mendes	Lani	M.	10

EMPLOYEE(Number, LastName, FirstName, MiddleName, Store)

STORE		
<u>Number</u>	Name	Address
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	North Dakota

STORE (Number, Name, Address)

Figura 28: Relações achadas no Employee works in a store.

5.2.2. Representando uma relação Binária M:N

O relacionamento de duas entidades se torna uma relação. A chave primária de uma relação é a combinação das chaves primárias das duas entidades. Qualquer atributo associado com o relacionamento se torna um atributo da relação. No exemplo da Figura 29, o relacionamento sells (Vende) torna-se uma relação chamada INVENTORY, e sua chave primária é a chave composta a qual é uma combinação com as chaves primárias de STORE e ITEM. Count e Operational Level são atributos de INVENTORY. O resultado das relações são mostrados na Figura 30.

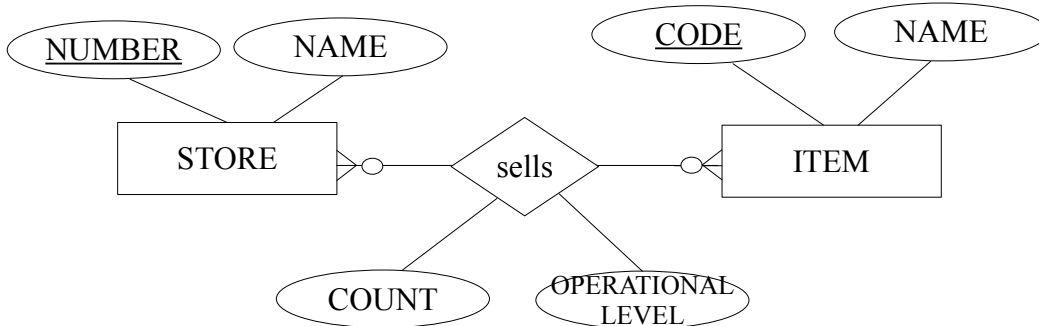


Figura 29: DER Store's inventory

STORE		
<u>Number</u>	Name	Address
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	North Dakota

STORE (Number, Name, Address)

ITEM	
<u>Code</u>	Description
1001	Wheat Germs
1002	Tarragon
1003	Yacon
1004	Thyme
1005	Bay Leaves

ITEM(Code, Description)

INVENTORY			
<u>Store_No</u>	<u>Item_Code</u>	Count	Operational Level
10	1001	2345	500
20	1001	4085	1000
10	1006	2115	300
20	1006	664	300

INVENTORY(Store_No, Code, Count, Operational Level)

Figura 30: Relacionamento Store's Inventory

5.2.3. Representando Relacionamento Unário

Um tipo de entidade é modelada como um relacionamento. A chave primária da relação é a chave primária da entidade. Uma chave estrangeira é adicionada à relação que referencia os valores da chave primária da mesma relação(chave estrangeira recursiva). No exemplo mostrado na Figura 31, um EMPLOYEE está sendo gerenciado por um MANAGER que é também um EMPLOYEE. O atributo Manager da relação é, na verdade, o employee Number do manager. O resultado das relações são mostrados na Figura 32.

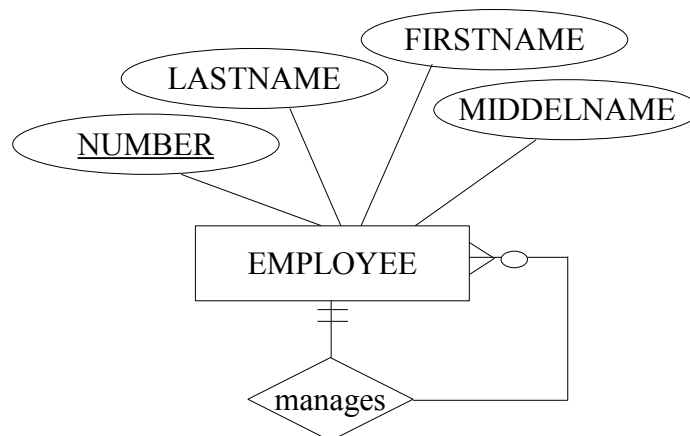


Figura 31: DER Employee is managed by a manager

EMPLOYEE					
<u>Number</u>	LastName	FirstName	MiddleName	Store	Manager
5000	Stone	Benjamin	J.	10	NULL
5001	Cruz	Juan	M.	10	5001
5002	Enriquez	Sheila	S.P.	10	5001
5003	Choo	James	Y.	10	5001
5004	Mendes	Lani	M	10	5001

EMPLOYEE(Number, LastName, FirstName, MiddleName, Store, Manager)

Figura 32: Relações achadas em Employee is managed by manager.

Um outro exemplo é mostrado na Figura 33 que mostra um relacionamento unário de muitos-para-muitos. Como revisão, o relacionamento se torna a relação. O resultado do relacionamento se torna uma relação COMPOSTA. Neste exemplo, um train consiste em panels, wheels e nails. A chave primária da COMPOSIÇÃO consiste no item **NUMBER** (Item_No), e a parte **NUMBER** (Component_No) do item.

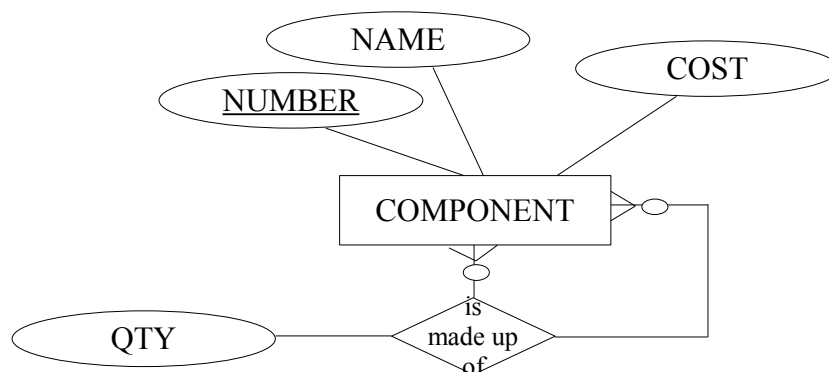


Figura 33: Componentes consistem de pequenos compoentes ERD

Component		
<u>Number</u>	Name	Cost
1001	Toy Train	Php900,00
1002	Panel of Train	Php50,00
1003	Train Wheels	Php20,00
1004	Nails	Php1,00

COMPONENT (Number, Name Cost)

Composition

Item_Number	Component_No	Qty
1001	1002	7
1001	1003	4
1001	1004	20

COMPOSITION(Item_No, Component_No, Qty)

Figure 34: Relations found in Components consists of smaller components.

5.2.4. Representando a Relação Is-A (Classe/Subclasse)

Os dados do modelo relacional não suporta diretamente relacionamento classe / subclasse. No entanto, existem diversas estratégias que podem se utilizar nos projetos. As estratégias que podem ser empregadas são as seguintes:

1. Criar uma relação separada para a classe e para cada uma das subclasse
2. A entidade ou relação ser constituída apenas para a classe dos atributos que são comuns a todos da subclasse
3. A entidade para cada subclasse conter apenas a sua chave primária e as colunas únicas da subclasse
4. As chaves primárias da classe e de cada uma das subclasses serem do mesmo domínio

Como um exemplo, considere-se a relação Is-A de EMPLOYEE na Figura 38. Atributos comuns a todas as subclasse de trabalhadores estão localizados na superclasse EMPLOYEE. Atributos específicos para a subclasse são encontradas na relação da subclasse. As relações são mostradas na Figura 36.

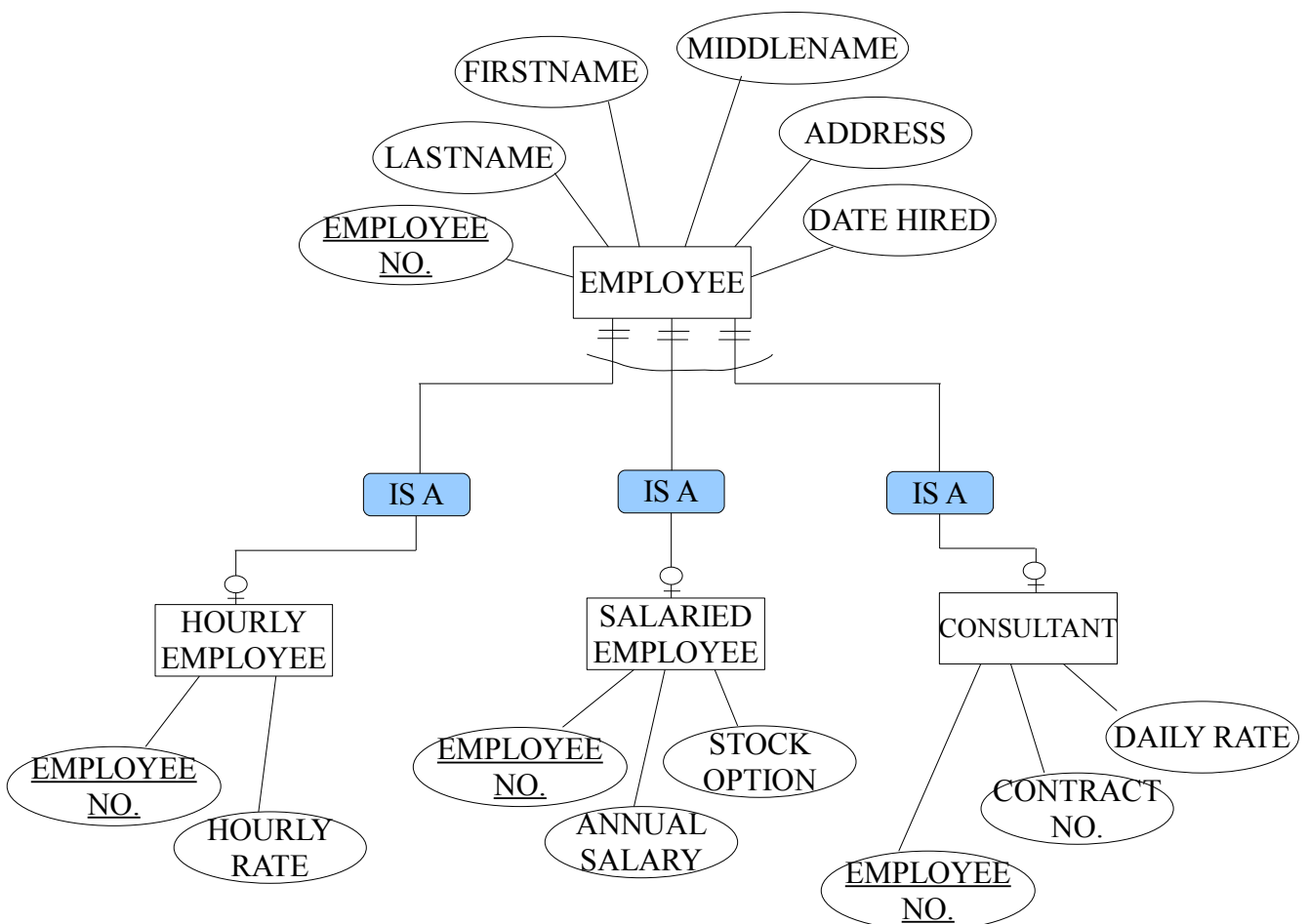


Figura 35: Categorização Employee ERD

EMPLOYEE

<u>Number</u>	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
5000	Stone	Benjamin	J.	10	NULL	12/25/1997
5001	Cruz	Juan	M.	10	5001	01/10/1998
5002	Enriquez	Sheila	S.P.	10	5001	01/10/1998
5003	Choo	James	Y.	10	5001	01/10/1998
5004	Mendes	Lani	M	10	5001	01/10/1998

EMPLOYEE(Number, LastName, FirstName, MiddleName, Store, Date_Hired)

HOURLY_EMPLOYEE

<u>Number</u>	Hourly_Rate
5002	Php100,00
5003	Php200,00

HOURLY_EMPLOYEE(Number, Hourly_Rate)

SALARIED EMPLOYEE

<u>Number</u>	Annual_Salary	Stock_Option
5001	Php345.000,00	N
5005	Php330.000,00	N

SALARIED_EMPLOYEE(Number, Annual_Salary, Stock_Option)

CONSULTANT

<u>Number</u>	Contract_No	Daily_Rate
5013	AH0001-10001	Php1.000,00

CONSULTANT(Number, Contract_No, Daily_Rate)

Figura 36: Categorização das Relações de Employee

5.2.5. PASSO 3: Normalizar as Relações

Após representando o ERD para as relações, o próximo passo é normalizar as relações. Normalizando-se a TNF forma usualmente suficiente. No entanto, se existe a necessidade de normalizar ainda mais, continuar até chegar à quinta forma normal. A lista completa de relações normalizadas (sem fusão) da Organic Shop é mostrado na Figura 37.

EMPLOYEE

<u>Number</u>	LastName	FirstName	MiddleName	<u>Store</u>
5000	Stone	Benjamin	J.	10
5001	Cruz	Juan	M.	10
5002	Enriquez	Sheila	S.P.	10
5003	Choo	James	Y.	10
5004	Mendes	Lani	M.	10

STORE

<u>Number</u>	Name	Address
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	North Dakota

ITEM

<u>Code</u>	Description
1001	Wheat Germs
1002	White Peppers
1003	Iodized Salts
1004	Oregano

INVENTORY

<u>Store_No</u>	<u>Item_Code</u>	Quantity	Operational_Level
10	1001	50	20
20	1001	2300	500
10	1004	4500	100
20	1004	90	100

EMPLOYEE

<u>Number</u>	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
5000	Stone	Benjamin	J.	10	NULL	12/25/1997
5001	Cruz	Juan	M.	10	5001	01/10/1998
5002	Enriquez	Sheila	S.P.	10	5001	01/10/1998
5003	Choo	James	Y.	10	5001	01/10/1998
5004	Mendes	Lani	M	10	5001	01/10/1998

HOURLY_EMPLOYEE

<u>Number</u>	Hourly_Rate
5002	Php100,00
5003	Php200,00

SALARIED EMPLOYEE

<u>Number</u>	Annual_Salary	Stock_Option
5001	Php345.000,00	N
5005	Php330.000,00	N

CONSULTANT

<u>Number</u>	Contract_No	Daily_Rate
5013	AH0001-10001	Php1.000,00

Figura 37: Normalizando as Relações de Organic Shop

5.2.6. PASSO 4: Juntando as Relações

Após normalizado, é necessário fundir as relações que se referem à mesma entidade. No entanto, deve-se verificar se existem problemas de integração, tais como as seguintes:

1. **Sinônimos.** Eles são dois atributos que têm nomes diferentes, mas o mesmo significado. Ao fundir as relações que contêm sinônimos, você deverá obter acordo (se possível) de usuários em nomes únicos e padronizados, para o atributo afim de eliminar sinônimos. No exemplo, o EMPLOYEE utiliza relação Number e Employee_no como a chave primária. Ambos têm o mesmo significado. Neste caso, escolhe-se um nome para o atributo; iremos usar Number.
2. **Homônimos.** Um único atributo pode ter mais do que um significado. Para resolver o conflito, é necessário criar novos nomes de atributos.
3. **Dependência de Transitividade.** Assegurar que, quando as relações são fundidos a transitividade das dependências não ocorram novamente.
4. **Classe/Subclasse (Is-A).** Se uma relação não parece correta, ou seja, falta determinados atributos, verifica-se se é uma subclasse de outra classe.

No exemplo, precisamos de fundir a relação em EMPLOYEE. A seguir temos o último conjunto de relações.

EMPLOYEE

<u>Number</u>	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
5001	Cruz	Juan	Martinez	10	5000	Dec. 25 2005
5002	Enriquez	Sheila	San Pedro	10	5001	Jun. 04 1998
5003	Ferrer	Grace	Atienza	20	5000	Jan. 16 1997

STORE

<u>Number</u>	Name	Address
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	North Dakota

ITEM

<u>Code</u>	Description
1001	Wheat Germs
1002	White Peppers
1003	Iodized Salts
1004	Oregano

INVENTORY

<u>Store_No</u>	<u>Item_Code</u>	Count	Operational Level
10	1001	50	20
20	1001	2300	500
10	1004	4500	100
20	1004	90	100

HOURLY_EMPLOYEE

<u>Number</u>	Hourly_Rate
10002	Php250,00
10005	Php250,00

SALARIED EMPLOYEE

<u>Number</u>	Annual_Salary	Stock_Option
10001	Php345.000,00	Yes
10006	Php546.000,00	Yes

CONSULTANT

<u>Number</u>	Contract_No	Daily_Rate
10004	AH0001-10001	Php1.000,00

Figura 38: O Projeto Lógico do Banco de Dados de Organic Shop

6. Exercícios

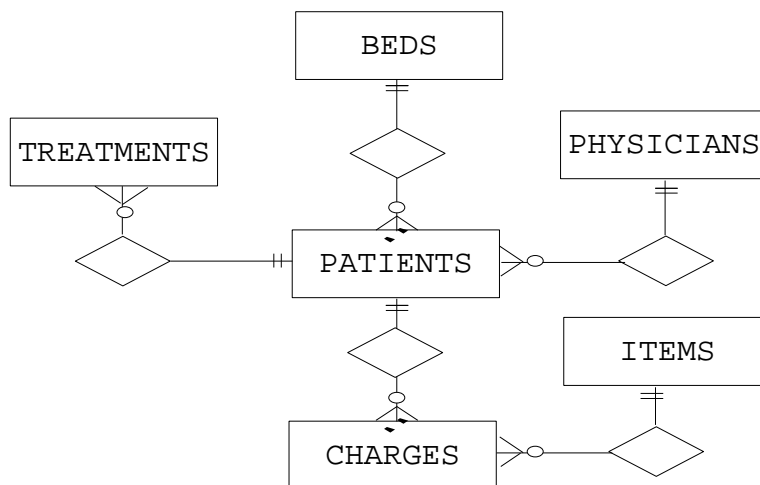
6.1. Normalizar

Normalizar de forma que o usuário visualize o conjunto de relações na 3ª Forma Normal (TNF).

CUSTOMER SERVICE REPORT						
Customer ID: SMI47 NOV02		Customer Name: Smith, J.		Service	Depot:	
		Address: 49 West Triangle Road Palmeras III Novaliches Quezon City		349 Rizal Ave., Pepsi Bldg. Novaliches		
				Service Visits		
Contract No.	Contract Date	Monthly Payment		Date	Job Number	
27015	27 Mar 1992	345.50		22 Aug 1992	34/3290	
				18 Mar 1993	34/4321	
24992	20 May 1992	1,345.00		13 Jul 1992	34/1766	
				18 Dec 1992	34/2541	
				06 Aug 1993	34/3115	

6.2. ERD

Transformar o seguinte ERD para uma relação bem estruturada. Atributos não são incluídos no diagrama afim de manter clareza. No entanto, os atributos são listados após o diagrama. Assegurar que as relações serão normalizadas.



Os atributos da entidade são os seguintes tipos:

- Bed
 - Bed Number
 - Bed Type
- Patients
 - Patient Number
 - Patient Name (Last Name, First Name, Middle Name)
 - Address
 - Telephone or Mobile Number
- Treatments
 - Treatment Number
 - Diagnosis
 - Treatment Description
- Physician
 - Physician Number

- Physician Name (Last Name, First Name, Middle Name)
 - Clinic Address
 - Telephone or Mobile Number
- Item
 - Item Number
 - Description
- Charges
 - Date charges are applied
 - Amount

6.3. *Diagrama de Entidade e Relacionamento*

- Transformar o diagrama entidade-relacionamento desenvolvido a partir do problema 1 no capítulo 2 (The Apartment Manager) em um conjunto de relações normalizadas.
- Transformar o diagrama entidade-relacionamento desenvolvido a partir do problema 2 no capítulo 2 (The Electronic Shop Manager) em um conjunto de

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.

Banco do Brasil

Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec

Suporte e apoio financeiro e logístico a todo o processo.

Borland

Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB

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

Módulo 9

Banco de Dados



Lição 4

Projeto Físico do Banco de Dados

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

1. Objetivos

Nesta lição, discutiremos o projeto físico do banco de dados para bancos de dados relacionais. Nesse capítulo, iremos discutir o modelo físico da base de dados a partir da base de dados lógica. Regras de negócio e restrições de integridade que inclui o tópico de restrições de domínio, integridade da entidade, integridade referencial e operações de gatilho. Quando estamos executando a análise de volume de dados, se estima o tamanho da base de dados que é usado para selecionar os dispositivos de armazenamento físico e estimar o custo de armazenamento.

Realizarmos uma análise de uso e estimar os caminhos ou padrões usados para selecionar a organização de arquivos e métodos de acesso para planejar o uso de índices e uma estratégia de distribuição dos dados.

A memória principal dos computadores é inadequada para o armazenamento da base de dados. Dispositivos secundários de armazenagem como discos rígidos e discos ópticos disponibilizam um meio para armazenar fisicamente a base de dados. A organização de arquivos e métodos de acesso são discutidos para entender como a base de dados são armazenadas fisicamente no armazenamento secundário. Para ajudar na eficiência do retorno de linhas na base de dados, índices foram discutidos, incluindo os diferentes tipos: nomeados, índices seqüenciais, não-seqüenciais, índice em cluster, índices em multi-níveis e árvores B/B+. Mostrar o conceito de desnormalização para otimizar certos processamentos de base de dados.

Ao final desta lição, o estudante será capaz de:

- Realizar o projeto físico do banco dados
- Conhecer as regras de negócio e restrições de integridade
- Formular o volume de dados e análise de uso
- Organizar os arquivos e conhecer os métodos de acesso
- Saber mais a respeito dos índices

2. Projeto Físico do Banco de Dados

É o processo de mapear as estruturas lógicas do banco de dados desenvolvidas nos estágios anteriores em um modelo interno ou um conjunto de estruturas físicas do banco de dados. Deve ser feito com cuidado já que as decisões tomadas neste estágio têm maior impacto na acessibilidade dos dados, tempo de resposta, segurança, uso amigável e fatores similares. O maior objetivo é implementar o banco de dados como um conjunto de registros armazenados, arquivos, índices e outras estruturas de dados que irão oferecer a performance adequada e assegurar a integridade do banco de dados, segurança e recuperação.

A fase anterior ao projeto físico, especificamente o projeto lógico do banco de dados, é altamente independente dos detalhes de implementação; sem funcionalidades específicas dos SGBD's e programas de aplicação alvos.

A saída do projeto lógico do banco de dados, por exemplo, o esquema relacional e o dicionário de dados são as fontes para o processo de projeto físico. Eles oferecem ao projetista de banco de dados uma maneira de fazer negociações que são muito importantes para um projeto de banco de dados eficiente.

O projeto lógico do banco de dados está relacionado com o *quê*; projeto físico do banco de dados está relacionado com o *como*. Ele requer habilidades que são geralmente encontradas em pessoas diferentes. Ele deve saber como o sistema de computador onde reside o SGBD opera e deve estar completamente consciente da funcionalidade do SGBD alvo.

As Três Maiores Entradas para o Projeto Físico do Banco de Dados:

1. Estrutura lógica do banco de dados (Esquema Relacional)
2. Requisitos de processamento do usuário que foram identificados durante a definição dos requisitos, incluindo o tamanho e a frequência de utilização do banco de dados.
3. Características do sistema de gerenciamento do banco de dados (SGBD) e outros componentes do ambiente de operação do computador.

O primeiro passo para transformar um projeto lógico de banco de dados em seu projeto físico de banco de dados envolve a tradução das relações derivadas do processo do projeto lógico do banco de dados em uma forma que possa ser implementada no SGBD relacional alvo. Isto requer conhecimento das funcionalidades oferecidas pelo SGBD alvo tais como:

- Se o sistema suporta a definição de chaves primárias, chaves estrangeiras e chaves alternativas
- Se o sistema suporta a definição de dados obrigatórios
- Se o sistema suporta a definição de domínios
- Se o sistema suporta a definição de restrições de negócio
- Como criar as relações base

Os objetivos de se definir a especificação no SGBD alvo é produzir um esquema básico de trabalho do banco de dados relacional das estruturas do projeto lógico do banco de dados que envolve decisões em como representar as relações base e projetar as restrições de negócio para o SGBD alvo. Para especificar o projeto físico das tabelas, precisamos considerar o seguinte:

- Regras de Negócio e Restrições de Integridade
- Volume de Dados e Análise de Uso
- Estratégias de Distribuição de Dados
- Organização do Arquivo e Métodos de Acesso ao Arquivo
- Índices
- Normalização

3. Regras de Negócio ou Restrições de Integridade

Regras de Negócio ou **Restrições de Integridade** são especificações que preservam a integridade do modelo lógico de dados. O termo *regras de negócio* é geralmente usado no contexto da fase de análise do banco de dados enquanto o termo *restrições de integridade* é usado no contexto da fase de projeto. Na fase de análise, o modelo conceitual ou Modelo ER está primariamente interessado na estrutura dos dados ao invés de expressar as regras de negócio. Entretanto, se uma regra de negócio é descoberta, ela deve ser documentada. Nas próximas seções do capítulo, o termo restrições de integridade será usado.

Na maioria dos sistemas de gerenciamento de banco de dados, eles incorporam as restrições de integridade dentro do escopo do sistema de banco de dados em vez de dentro dos programas de aplicação ou operadores humanos. Algumas das vantagens de se colocar as restrições de integridade nos sistemas de gerenciamento de banco de dados são:

1. Oferece desenvolvimento mais rápido de aplicações com poucos erros.
2. Reduz o esforço e o custo de manutenções.
3. Oferece resposta mais rápida às mudanças de negócio.
4. Facilita o envolvimento do usuário final no desenvolvimento de novos sistemas e na manipulação dos dados.
5. Oferece aplicações consistentes das restrições de integridade.
6. Reduz o tempo e esforço necessário para treinar programadores de aplicação.
7. Promove facilidade de utilização de um banco de dados.

Três Categorias de Restrições de Integridade:

1. Restrições de Domínio
2. Integridade da Entidade
3. Integridade Referencial
4. Operações de Gatilhos (*Triggers*)

A entrada básica pra o processo de projeto físico é a relação ou tabela. Como revisão, uma **tabela** é uma representação bidimensional de dados ou entidades consistindo de uma ou mais colunas, e zero ou mais linhas. Figura 1 mostra um exemplo de uma relação ou tabela com os seus componentes correspondentes. Esta é a tabela ou relação `STORE`.

Nome da Tabela	STORE		
Atributos			
Amostra de Valores			
	Number	Name	Address
	10	GangStore in Alabama	Alabama
	20	GangStore in West Virginia	West Virginia
	30	GangStore in North Dakota	North Dakota

Figura 1: Tabela ou Relação Store

Na nomeação de tabelas e colunas, as seguintes regras devem ser observadas:

- Os nomes das tabelas devem ser únicos. Dentro do modelo de dados, não deve haver duas tabelas com o mesmo nome.
- Os nomes das colunas devem ser únicos dentro de uma tabela. Entretanto, colunas em tabelas diferentes podem compartilhar o mesmo nome.
- As linhas devem ser únicas. As linhas consideradas em sua totalidade devem ser distintas umas das outras.

3.1. Restrições de Domínio

Um entendimento de domínios é importante na definição de restrições de integridade. Um **domínio** é um conjunto de todos os tipos de dados e série de valores que atributos podem assumir. Tipicamente, especifica:

- significado
- tipo do dado
- valores permitidos
 - unicidade
 - suporte a nulo
 - intervalo
- formato
 - tamanho
 - projeto do código
- série

As vantagens do uso de domínios são:

1. Valida os valores de um atributo.
2. Ajuda a poupar esforço na descrição das características do atributo.

Exemplos de restrições de domínio são dadas na Figura 2. Um `ACCOUNT` tem alguns `WITHDRAWALS`.

ACCOUNT

ACCOUNT NO.	...	BALANCE
SAR-1034	...	12987,34
SAR-1125	...	3000,00
CAR-1256	...	5126,14
CAR-3756	...	9834,33

WITHDRAWAL

ACCOUNT NO.	TRANSACTION DATE	...	AMOUNT
SAR-1034	25/03/01 15:00	...	500,00
SAR-1034	01/04/01 11:30	...	1500,00
CAR-3756	22/01/01 09:30	...	3000,00

Atributo: ACCOUNT NO.

Significado: Número da conta do cliente no banco

Tipo do Dado: Caracter

Formato: AAA-AAAA

Unicidade: Deve ser único

Suporte a Nulo: Não-nulo

Atributo: AMOUNT

Significado: Quantidade do Saque

Tipo do Dado: 2 Casas Decimais

Intervalo: 0-P10,000

Unicidade: Não-único

Suporte a Nulo: Não-nulo

Figura 2: Exemplos de Restrições de Domínio

Tipos de Dados

Para cada coluna identificada, define os tipos de dados válidos. Podemos usar tipos de dados de diferentes linguagens de programação para representar os tipos de dados válidos que uma coluna pode ter. Neste curso, usaremos os tipos de dados JavaDB que são compatíveis com o SQL.

Tipos de Dados Numéricos

Tipos Numéricos incluem os seguintes tipos, que oferecem armazenamento de tamanhos variados:

- Numéricos Inteiros
 - SMALLINT
 - Sintaxe: SMALLINT
 - Oferece 2 bytes de armazenamento.
 - O intervalo é de -32.768 a 32.767.
 - INTEGER
 - Sintaxe: {INTEGER | INT}
 - Oferece 4 bytes de armazenamento para valores inteiros.
 - O intervalo é de -2147483648 a 2147483647.
 - BIGINT
 - Sintaxe: BIGINT
 - Oferece 8 bytes de armazenamento para valores inteiros.
 - O intervalo é de -9223372036854775808 a 9223372036854775807.
- Numéricos Aproximados ou de Ponto-Flutuante
 - REAL
 - Sintaxe: REAL

- Oferece 4 bytes de armazenamento para números usando a notação de ponto-flutuante IEEE.
- Constantes numéricas de ponto-flutuante são limitadas a 30 caracteres de comprimento.
- Limitação dos seus intervalos:
 - Menor valor REAL: -3.402E+38
 - Maior valor REAL: 3.402E+38
 - Menor valor positivo REAL: 1.175E-37
 - Maior valor negativo REAL: -1.175E-37
- DOUBLE PRECISION
 - Sintaxe: {DOUBLE PRECISION | DOUBLE}
 - Oferece 8 bytes de armazenamento para números usando a notação de ponto-flutuante IEEE.
 - É o sinônimo de DOUBLE.
 - Constantes numéricas de ponto-flutuante são limitadas a 30 caracteres de comprimento.
 - Limitação dos seus intervalos:
 - Menor valor DOUBLE: -1.79769E+308
 - Maior valor DOUBLE: 1.79769+308
 - Menor valor positivo DOUBLE: 2.225E-307
 - Maior valor negativo DOUBLE: -2.225E-307
- Número Exato
 - DECIMAL (armazenamento baseado na precisão)
 - Sintaxe: {DECIMAL | DEC}[(precisão [,escala])]
 - Oferece um numérico exato no qual a precisão e a escala podem ser atribuídos arbitrariamente.
 - A quantidade de armazenamento necessário será baseado na precisão.
 - A precisão definirá o número total de dígitos, tanto para a esquerda quanto para a direita do ponto decimal; a escala definirá o número de dígitos do componente fracionário.
 - A precisão deve estar entre 1 e 31; a escala deve ser menor ou igual à precisão.

Tipos de Dados Caracteres

Tipos Caracteres incluem os seguintes:

- Tipos Caractere de Tamanho-fixado
 - CHAR
 - Sintaxe: CHAR[ACTER][(tamanho)]
 - Oferece armazenamento de strings de tamanho fixo.
 - O tamanho é um inteiro constante sem sinal. O tamanho padrão é 1.
 - Espaços completam um valor de string menor que o tamanho esperado. Espaços remanescentes são truncados se você tentar inserir um string de tamanho superior. Ocorre erro se o string resultante terminar sem espaços e ainda assim for muito comprido.

- CHAR FOR BIT DATA
 - Sintaxe: {CHAR | CHARACTER}[(tamanho)] FOR BIT DATA
 - Permite armazenar strings de bytes para um tamanho específico.
 - É útil para dados não estruturados onde strings de caracteres não são apropriados.
 - O tamanho é um literal inteiro sem sinal designando o tamanho em bytes.
 - O tamanho padrão é 1; o tamanho máximo é 254 bytes.
- Tipos Caractere de Tamanho-variável
 - VARCHAR
 - Sintaxe: {VARCHAR | CHAR VARYING | CHARACTER VARYING} (tamanho)
 - Oferece armazenamento para strings de tamanho variável.
 - O tamanho é um inteiro constante sem sinal, e não deve ser maior que a restrição do inteiro usado para especificar o tamanho.
 - O tamanho máximo é 32.672 caracteres.
 - Espaços não são incluídos se o valor é menor que o tamanho.
 - Espaços finais são truncados para o tamanho da coluna. Se o string resultante continuar grande, um erro ocorre.
 - LONG VARCHAR
 - Sintaxe: LONG VARCHAR
 - Permite armazenamento de caracteres de um tamanho máximo de 32.700 caracteres.
 - É similar ao VARCHAR exceto que o número máximo de caracteres não é especificado.
 - VARCHAR FOR BIT DATA
 - Sintaxe: {VARCHAR | CHAR VARYING | CHARACTER VARYING} (tamanho) FOR BIT DATA
 - Permite armazenar strings binários menores ou iguais ao tamanho especificado.
 - É útil para dados não estruturados onde strings de caracteres não são apropriados (tais como nas imagens).
 - O tamanho é um inteiro constante sem sinal definindo o tamanho em bytes.
 - Não tem tamanho padrão ao contrário do CHAR FOR BIT DATA.
 - O tamanho máximo do tamanho é 32.672 bytes.
 - LONG VARCHAR BIT DATA
 - Sintaxe: LONG VARCHAR FOR BIT DATA
 - Permite o armazenamento de strings de bits até 32.700 bytes.
 - É idêntico ao VARCHAR FOR BIT DATA exceto que o tamanho máximo não está especificado.

Tipos de Dados de Data e Hora

- DATE
 - Sintaxe: DATE
 - Oferece armazenamento de um ano-mês-dia em um intervalo suportado pelo java.sql.Date.

- Suporta os seguintes formatos:
 - aaaa-mm-dd
 - mm/dd/aaaa
 - dd.mm.aaaa
- Não deve ser misturado com TIME e TIMESTAMP em uma expressão.
- TIME
 - Sintaxe: TIME
 - Oferece armazenamento de um valor hora-do-dia.
 - Suporta os seguintes formatos:
 - hh:mm[:ss]
 - hh.mm[:ss]
 - hh[:mm] {AM | PM}
- TIMESTAMP
 - Sintaxe: TIMESTAMP
 - Armazena um valor combinado de DATE e TIME.
 - Permite fracionamento de segundos até nove dígitos.
 - Suporta os seguintes formatos:
 - aaaa-mm-dd hh:mm:ss[.nnnn]
 - aaaa-mm-dd-hh.mm.ss[.nnnn]

Tipos de Dados para Grandes Objetos

- CLOB
 - Sintaxe: {CLOB | CHARACTER LARGE OBJECT} [(tamanho [{K | M | G}])]
 - É conhecido como objeto de caractere grande com um valor até 2.147.483.647 de comprimento.
 - É usado para armazenar dados baseados nos caracteres unicode, tais como grandes documentos em qualquer conjunto de caracteres.
 - O tamanho é dado em número de caracteres em ambos CLOB, a menos que um dos sufixos K, M ou G é dado, referindo-se ao múltiplos de 1024, 1024*1024, 1024*1024*1024, respectivamente.
- BLOB
 - Sintaxe: {BLOB | BINARY LARGE OBJECT} [(tamanho [{K|M|G}])]
 - É um string binário de tamanho variável que pode ser de até 2.147.483.647 caracteres de comprimento.
 - Assim como outros tipos binários, não está associado a uma página de código; não armazena dados caracteres.
 - O tamanho é dado em bytes a menos que um dos sufixos, K, M or G, é dados, referindo-se aos múltiplos de 1024, 1024*1024, 1024*1024*1024, respectivamente.

Outros Tipos de Dados

- XML
 - Sintaxe: XML

- É usado para documentos Extensible Markup Language (XML).
- É usado:
 - para armazenar documentos XML que estão de acordo com a definição SQL/XML de um bem formado valor XML(DOCUMENT(ANY)).
 - Provisoriamente para valores XML(SEQUENCE), que podem não ser um valor bem formado XML(DOCUMENT(ANY)).

STORE

Number	Name	Address
SMALLINT	VARCHAR(250)	VARCHAR(250)
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	NorthDakota

EMPLOYEE

Number	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
SMALLINT	VARCHAR(100)	VARCHAR(100)	CHAR(4)	SMALLINT	SMALLINT	DATE
5001	Cruz	Juan	M.	10	5000	25 de Dez de 2005
5002	Enriquez	Sheila	S.P.	10	5001	4 de Jun de 2008
5003	Choo	James	Y.	10	5000	16 de Jan de 1997
5004	Mendes	Lani	M.	10	5001	4 de Jan de 1998

Figura 3: Especificando Tipos de Dados

Valores Permitidos

O que deve ser considerado a seguir são os valores permitidos para cada coluna. Algumas considerações são os valores nulos, valores duplicados, valores modificáveis, chaves e dados derivados ou calculados.

Valores Nulos

Um valo **nulo** é um valor que está faltando ou desconhecido em uma coluna de uma tabela. Eles não são o mesmo que brancos. Dois brancos são considerados iguais em valor; a equivalência de dois valores nulos é indeterminado. Embora, eles possam aparecer na tabela como brancos.

Nulos não são iguais a zeros. A maioria das operações aritméticas podem ser realizadas nos valores zero; nulos devem ser excluídos das manipulações matemáticas.

NN significa nulo não-permitido. Para indicar que nulos não são permitidos na coluna de uma tabela, coloque letras NN diretamente abaixo do devido cabeçalho da coluna.

STORE

Number	Name	Address
NN	NN	
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	NorthDakota

EMPLOYEE

Number	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
NN	NN	NN		NN		NN
5001	Cruz	Juan	Martinez	10	5000	25 de Dez de 2005
5002	Enriquez	Sheila	San Pedro	10	5001	4 de Jun de 2008
5003	Ferrer	Grace	Atienza	20	5000	16 de Jan de 1997
5004	Franz	Jane	Solamo	20	5003	4 de Jan de 1998

Figura 4: Especificando Valores Não-nulos

Figura 4 mostra um exemplo de especificação de valores de não-nulos nas tabelas STORE e EMPLOYEE. As colunas Number e Name da tabela STORE são não-nulas. Similarmente, as colunas Number, LastName, FirstName, Store e Date_Hired da tabela EMPLOYEE são não-nulas.

Valores Duplicados

Um **valor duplicado** é um valor em uma coluna de uma tabela que exatamente se iguala a algum outro valor naquela mesma coluna.

- **ND** significa nenhuma duplicação permitida. Enquanto a duplicação de uma linha inteira não é permitida, valores duplicados em uma coluna particular de tabelas são comuns.
- Quando a combinação de valores de certas colunas da tabela precisam ser únicas, podemos usar numeração nas marcas **ND** para representar o grupo de colunas. Para mostrar um exemplo, na tabela inventory, a combinação dos valores das colunas store_no e item_code deve ser única na tabela. **ND1** representa o agrupamento.

INVENTORY

Store_no	Item_code	Quantity	Op_Level
NN,ND1	NN,ND1	NN	NN
10	1001	2345	500
10	1006	3245	100
20	1001	456	100

EMPLOYEE

Number	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
NN,ND	NN	NN		NN		NN
5001	Cruz	Juan	Martinez	10	5000	25 de Dez de 2005
5002	Enriquez	Sheila	San Pedro	10	5001	4 de Jun de 2008
5003	Ferrer	Grace	Atienza	20	5000	16 de Jan de 1997
5004	Franz	Jane	Solamo	20	5003	4 de Jan de 1998

Figura 5: Especificando Valores Não-Duplicados

Figura 5 mostra como especificar valores não duplicados. Para a tabela STORE, os valores combinados de Number e Name devem ser únicos. Para a tabela EMPLOYEE, o Number deve ser único.

Valores Alteráveis

Um **valor alterável** é um valor em uma tabela que pode variar com o tempo. A maioria dos valores na maioria das tabelas são alteráveis.

- **NC** significa que nenhuma alteração é permitida.

STORE

Number	Name	Address
NN,ND1,NC	NN,ND1	
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	NorthDakota

EMPLOYEE

Number	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
NN,ND,NC	NN	NN		NN		NN,NC
5001	Cruz	Juan	Martinez	10	5000	25 de Dez de 2005
5002	Enriquez	Sheila	San Pedro	10	5001	4 de Jun de 2008
5003	Ferrer	Grace	Atienza	20	5000	16 de Jan de 1997
5004	Franz	Jane	Solamo	20	5003	4 de Jan de 1998

Figura 6: Especificando Valores Não-alteráveis

Figura 6 apresenta um exemplo de especificação de valores não alteráveis. Na a tabela STORE (LOJA), o Número (Number) e o Nome (Name) não são permitidos serem alterados uma vez inseridos na tabela. Na tabela EMPLOYEE (FUNCIONÁRIOS), o Número (Number) e a Data de Admissão (Date_Hired) não são alteráveis.

Chaves

Dois tipos de chaves necessitam serem especificadas. Nominalmente, a chave primária e a chave estrangeira.

A **chave primária** de uma tabela é a coluna ou grupo de colunas que possuem valores únicos identificando cada linha da tabela. As regras devem ser observadas:

- Toda tabela deve ter uma chave primária.
- Cada linha de dados deve ser sempre identificável unicamente.
- Toda tabela deve ter apenas uma chave primária.
- PK significa chave primária; todas as chaves primárias obedecem as seguintes regras:
 - Valores da chave primária nunca devem ser nulos.
 - Valores da chave primária nunca devem ser duplicados.
 - Valores da chave primária nunca devem ser alteráveis.
- A chave primária pode ser determinada pelo sistema. **SA** significa determinada pelo sistema (system-assigned). Se o valor da chave primária é automaticamente gerada pelo sistema quando uma nova linha é adicionada, esses valores são ditos para serem determinados pelo sistema, e são marcados como PK,SA. Isto é verdadeiro até mesmo quando a determinação é feita manualmente mas de uma lista gerada por computador.
- A chave primária pode ser determinada pelo usuário. **UA** significa determinada pelo usuário. Se o valor de uma chave primária é especificada por usuários do sistema, tais valores são ditos como determinados pelo usuário e são marcados como PK,UA.
- Observe que as marcações NN, ND e NC já não são necessárias uma vez que estas restrições já estão incluídas na definição da chave primária.

A chave estrangeira é uma coluna ou grupo de colunas que são chaves primárias de outras tabelas.

- **FK** significa chave estrangeira (foreign key). Chaves estrangeiras de múltiplas colunas são numeradas como múltiplas colunas ND constraints.

STORE						
Number	Name			Address		
PK , UA , ND1	NN , ND1					
10	GangStore in Alabama			Alabama		
20	GangStore in West Virginia			West Virginia		
30	GangStore in North Dakota			NorthDakota		

EMPLOYEE						
Number	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
PK , UA	NN	NN		NN , FK		NN , NC
5001	Cruz	Juan	Martinez	10	5000	25 de Dez de 2005
5002	Enriquez	Sheila	San Pedro	10	5001	4 de Jun de 2008
5003	Ferrer	Grace	Atienza	20	5000	16 de Jan de 1997
5004	Franz	Jane	Solamo	20	5003	4 de Jan de 1998

Figura 7: Especificando chaves

Figura 7 apresenta como especificar chaves. Para a chave primária, o NN, ND e NC são removidos e substituídos por PK. Na tabela STORE, a chave primária é Number e é determinada pelo usuário. Na tabela EMPLOYEE, a chave primária é Number e é determinada pelo usuário. A coluna Store é uma chave estrangeira que referencia a chave primária da tabela STORE.

Derivado ou Dado Calculado

Dados derivados são dados que são calculados a partir de dados em um modelo definido em outro local. São dados redundantes que complicam a operação de atualização nas tabelas. Devem ser evitados e incluídos no modelo apenas quando graves problemas de performance ditar as suas necessidades.

- **DD** significa dados derivados.

Formatos e Projeto de Código

Códigos são um conjunto de números inteiros ou letras usadas para representar a descrição de um item de forma curta para um processamento eficiente. É usado para:

- Identificação e Recuperação
 - Número da Conta
 - Código Item
- Classificação
 - Tipo Conta
 - Status Conta
 - Tipo Demanda
- Eficiente classificação e indexação
 - Códigos curtos resultam em menor tempo de indexação e classificação
- Economia de custos na preparação dos dados
 - Nº Conta substituindo Nome do Cliente
 - Tipo de Conta ao invés de "Residencial" ou "Comercial"

Orientações para Projeto de Código

Algumas orientações para projetar códigos para os valores das colunas.

- Códigos que são:
 - Únicos
 - devem fornecer um valor de código na tabela
 - Numéricos
 - devem ser chaves fáceis de codificar
 - devem ser limitadas a 7-10 dígitos ou menos
 - Alfabéticos
 - devem ser fáceis de serem lembrados
 - devem ser limitadas de 5 a 7 letras ou menos
- Não misture Numéricos e Alfabéticos; exemplos:
 - X989W34H5 é horrível
 - PEC 490 é permitido
 - Códigos devem ser coerentes no formato
 - Para códigos de 4 dígitos atribua 1000 – 9999
- Não utilize baixos valores não significativos precedidos por zero como
 - 0001 pode ser escrito como 1
- Códigos que são protegidos por um dígito verificador:

- é utilizador para evitar erros
- Utilizado quando alta precisão é necessária

Tipos de Esquema de Códigos

A seguir estão alguns esquemas ou estratégias usadas para projetar o formato dos códigos:

1. Esquemas de Códigos Alfanuméricos

- Código Alabéticos de Tamanho Fixo
 - Códigos Monetários – Usado pelo Banco Mundial, FMI e IATA

Formato: XXX

 - 2 Primeiros Caracteres – Código do País
 - Últimos Caracteres – Código da Moeda

Exemplos:

PHP	Filipinas Peso
SGD	Singapura Dólar
HKD	Hong Kong Dólar
USD	US Dólar
JPY	Japão Yen

- Códigos Alfabéticos de Tamanho Fixo
 - Unidades de Medida – Usado pela Marinha US e Mattel

Formato: XX Código de duas letras

Exemplos:

MT	Tonelada Métrica
PC	Pedaço
ST	Conjunto
Li	Litro
KM	Quilometro

- Código Alfabético de Tamanho Variável
 - Unidades de Medida – Uso de Abreviações Aceitas Universalmente

Formato: XXX 1 a 3 códigos de caracteres

Exemplos:

SET	Conjunto
PCS	Pedaços
LBS	Libras
L	Litros
KM	Quilômetros
M	Metros

- Derivações Alfabéticas
 - Usadas em programas e abreviações

Formato: Um a sete caracteres

Exemplo:

Andrews		ANDRWS
Smith	SMTH	
Counter		CTR
Accumulator	AC	
Save Area		SA
Switch		SW ou SWC

2. Modo Combinado de Esquema de Codificação

- Códigos de Som ou Códigos Fonéticos

Formato: X999

- Peque a primeira letra
- Ignore todas as vogais A, E, I, O, U e as letras W, H, Y
- Troque as próximas três letras por:

- 1 B,F,P,V
- 2 C,G,J,K,Q,S,X,Z
- 3 D,T
- 4 L
- 5 M,N
- 6 R

Exemplo:

Diaz, Mario Avanceña
D 2 5 6
Uy, Whu Yu
U 0 0 0
Chua, Joy Uy
C 2 0 0

- Imposto Antigo do Nº da Conta (TAN)

Formato: X9999 – X999 – X – 9

Soundex

Aniversário

Sublinhado – Cartões Alegres

Note: X do Aniversário (I não é usado)

JAN	A	JUL	G
FEV	B	AGO	H
MAR	C	SET	J
ABR	D	OUT	K
MAi	E	NOV	L
JUN	F	DEZ	M

Exemplos:

1. Rizal, Joe Concepcion born Mar. 12, 1955

R242? - C1255 - ? - ?

pode ser R2428 – C1255 – A – 7

ou R2420 – C1255 – S – 0

- Codificando Nº da Placa

Formato: XXX 999

- Código de 3 letras – determinado pela região
 - N Reservado – veículos públicos
 - S – veículos governamentais
 - O I – não utilizados
- Series – 001 a 999

Exemplos:

- PEC 490 veículo privado em Metro Manila
- SCR 152 veículo governamental
 - NTX 442 veículo público

3. Esquemas Numéricos de Codificação

- Seqüência

- Normalmente utilizada quando não há código existente. Os valores a serem codificados são dispostos alfabeticamente e números consecutivos são atribuídos. Os números restantes são utilizados para qualquer novo nome.

100 – Aaron, James

102 – Abalos, Jane

:

465 – Mendoza, Estilito

466 – Mendoza, Fred

:

875 – Zapra, Jose

- Pré-alocação de um conjunto de números para representar um grupo de produtos ou informação

- 10000 – 19999

Galvanized Steel Sheet Metal

- 20000 – 29999

Steel Plates

- 30000 – 49999

Pre-fabricated Building Support

- 50000 – 79999

Subassemblies

- Codificação de Dígito Significante

- Certa porção do código representa alguma característica significativa da informação a ser codificada

9999

primeiro dígito Faixa de Voltagem

últimos 3 dígitos Watts

Examples:

- 2025 Luzes 240v 25w

- 2040 40w

- 2100 100w

- 1050 Luzes 110v 50w

- 1080 80w

- Código de Aparência

- Este tipo de código é semelhante ao código bloqueado mas tem mais sub-classificações no âmbito de cada faixa de código

NATO Número Armazenado

9999 99 999 9999

Grupo Classe Nação Item Número

- Exemplo de Código de Aparência

- Livro Geral de Codificação

9 9999 999

- 1 – Ativo

- 2 – Passivo

- 3 – Acionistas

- 4-5 – Salário

- 6-9 – Despesas

1XXX Ativo

1100 Caixa
 1110 Caixa em banco
 1111 Poupança
 1112 Far East Bank

4. Códigos Numéricos Protegidos

- Certos códigos requerem alta precisão na codificação para garantir que os valores estão corretos, um dígito verificador é adicionado ao código em ordem para proteger dos seguintes tipos de erros:

3 9 1 7 5	escrito como...
3 9 7 5	Omissão
3 9 1 7 5 2	Adição
8 9 1 7 5	Transcrição
9 3 1 7 5	Única Transcrição
3 1 9 5 7	Duplo Transcrição
3 2 9 6 7	Randômico

- A técnica mais comum utilizada é o Módulo Aritmético. Cada posição do dígito é multiplicado através de um peso pré-definido, os produtos resultantes são adicionados e a soma é dividida pelo módulo escolhido. O dígito verificador é calculado como o módulo menos o restante do processo de divisão.

Exemplo: Módulo 11

- No. Conta Base
 4 5 3 6 1
- Multiplique cada dígito escolhido pelo peso
 6 5 4 3 2
- Produtos
 24 25 12 18 2
- Somatório dos Produtos
 $24+25+12+18+2 = 81$
- Divida a soma por 11 e retire o resto da divisão
 $81/11 = 7$ restou 4
- Subtraia o resto de 11
 $11-4 = 7$
- O próprio numero verificador da conta é
 45 3 61 7

Exemplo: Módulo 10

- No. Conta Base
 4 5 3 6 1
- Multiplique cada dígito escolhido pelo peso
 2 0 2 0 2
- Produtos
 8 0 6 0 2
- Somatório dos Produtos
 $8+0+6+0+2 = 16$
- Divida a soma por 10 e retire o resto da divisão

$$16/10 = 1 \text{ restou } 6$$

- Subtraia o resto de 10

$$10-6 = 4$$

- O proprio numero verificador da conta é

4 5 3 6 1 7

- A porcentagem de erros detectados através dos módulos

As técnicas 11 e 10 :

MÓDULOS	10	11
■ Transcrição	100%	94%
■ Única Transcrição	100%	90%
■ Dupla Transcrição	100%	90%
■ Randômico	90%	90%
■ Substituição de número valido porém incorreto	0%	0%

Passos no Projeto de Código

1. Descreva a informação a ser codificada.
2. Identifique o propósito do código. Determine se há um código em uso.
3. Determine o número inicial de elementos abrangidos pelo código.
4. Extrapole o número esperado de itens cobertos para os próximos 3 a 6 anos.
5. Identifique a mídia em que cada código irá aparecer como o monitor do computador, formulários e/ou relatórios.
6. Identifique o desempenhos das tarefas (identificação, classificação...) através das pessoas que estão utilizando os códigos.
7. Determine requerimentos precisos, detecção de erros e correções. Isso irá determinar se um dígito verificador é requerido.
8. Determine se há uma limitação computacional imposta no código (número de dígitos significativos suportados através da linguagem ou hardware).
9. Projete a estrutura do código.
10. Construa o código.

Figura 8 apresenta a definição de cada coluna do *The Organic Shop* e como os formatos de códigos são especificados nas tabelas.

STORE		
Number	Name	Address
SMALLINT	VARCHAR(250)	VARCHAR(250)
99	X(250)	X(250)
PK, UA, ND1	NN, ND1	
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	NorthDakota

EMPLOYEE

Number	LastName	FirstName	MiddleName	Store	Manager	Date_Hired
SMALLINT	VARCHAR(100)	VARCHAR(100)	CHAR(4)	SMALLINT	SMALLINT	DATE
9999	X(100)	X(100)	X(100)	99	9999	Mon dd, yyyy
PK, UA	NN	NN		NN, FK		NN, NC
5001	Cruz	Juan	Martinez	10	5000	25 de Dez de 2005
5002	Enriquez	Sheila	San Pedro	10	5001	4 de Jun de 2008
5003	Ferrer	Grace	Atienza	20	5000	16 de Jan de 1997
5004	Franz	Jane	Solamo	20	5003	4 de Jan de 1998

ITEM

Code	Description
SMALLINT	VARCHAR(250)
99	X(250)
PK, UA	NN
1001	Wheat Germs
1002	White Peppers
1003	Iodized Salts
1004	Oregano
1005	Basil Leaves

INVENTORY

Store_no	Item_code	Qty	Op_level
SMALLINT	SMALLINT	INTEGER	INTEGER
99	1XXX	99999	99999
PK1, FK	PK1, FK		
10	1001	50	20
20	1001	2300	500
10	1004	4500	100
20	1004	90	100

HORARIO_FUNCIONARIO

Numero	Horas_Trabalhadas
SMALLINT	DECIMAL
5XXX	999999,99
PK	
5001	250,00

SALARIO_FUNCIONARIO

Numero	Salario_Anual	Loja_opcional
SMALLINT	DECIMAL(30,2)	CHAR(1)
5XXX	999999,99	Y or N
PK		default='N'
5001	546000,00	N

CONSULTA

Numero	Num_Contrato	Comissao_Diaria
SMALLINT	VARCHAR(25)	DECIMAL(20,2)
5XXX	X(250)	999999,99
PK		
5005	AH0001-10001	750,00

Figura 8: Aparência de uma tabela de Banco de Dados de uma loja

3.2. Integridade da Entidade

Uma base de relacionamentos ou tabela base corresponde a uma entidade conceitual cujo tuplos ou filas são fisicamente armazenadas no banco de dados. A integridade da entidade significa que a base da relação da chave primária (quer sejam simples ou compostas) não pode ser nula. Por definição, uma chave primária é no mínimo um identificador utilizado para identificar linhas exclusivas. Isto significa que nenhum subconjunto da chave primária é suficiente para fornecer uma identificação única de linhas. Se *nulls* (nulo) foram autorizados, implica que nem todas as colunas são necessárias para estabelecer uma distinção entre as linhas, o que contradiz a definição da chave primária.

A integridade da entidade também é conhecido como **primary key constraint**.

3.3. Integridade Referencial

Integridade referencial define as *constraints* que abordam a validade de referências a uma tabela para alguma outra tabela ou tabelas no banco de dados. Diz respeito à forma como são definidas as chaves estrangeiras. Se existe uma chave estrangeira em uma tabela, quer o valor chave estrangeira deve corresponder a um valor de alguns candidatos-chave na sua própria tabela ou o valor chave estrangeira deve ser nulo.

Como um exemplo, considere FUNCIONÁRIO e LOJA na tabela da Figura 1. A coluna LOJA da tabela FUNCIONÁRIO é considerado uma chave estrangeira. Os valores encontrados são os mesmos da coluna NÚMERO da tabela LOJA. Pode-se dizer que o quadro é referenciar o trabalhador, e da tabela é referenciada da loja. Cada valor na coluna na chave estrangeira referenciando a tabela deve ser o mesmo valor na chave primária correspondente coluna da tabela referenciada, ou então, deve ser nulo.

Regras referenciais de integridade

1. Regra para inserir. Esta linha afirma que não deve ser inserido na mesma tabela de referenciamento a menos que já exista uma correspondente entrada na tabela referenciada. Considere o registro da Figura 2. Suponhamos que não haja registro em Loja Número 99 na tabela LOJA. Se essa linha é inserida, seria violar a inserção regra. A fim de permitir a inserção, quer atribuir um número que existe na tabela da loja ou atribuir um valor nulo (assumindo nulls são permitidos para esta coluna).

FUNCIONARIO	
Numero	5110
Sobrenome	Salazar
Nome	Ceasar
NomeMeio	Chan
Loja	99
Gerente	5001
Data_Inicio	15/02/07

Figura 9: Funcionario 5110 gravado

2. Regra para deletar. Indica que uma linha não deve ser suprimida a partir da tabela referenciada se há alguma correspondência linhas na tabela do referenciamento. Suponhamos que uma Loja pretende apagar as informações para a Loja Número 20 (GangStore na Virgínia Ocidental). Este pedido irá violar a regra de supressão. Há três coisas que se pode fazer:
 - **Restringir.** Fará com que o pedido de supressão não seja aceito ou negado. O sistema não permitirá a supressão da Loja número 20 na tabela LOJA, e apresentará um erro;
 - **Anular.** Significa que os valores da chave estrangeira da tabela de referenciamento sejam fixados a um valor nulo. No exemplo, todas as linhas na tabela o empregado trabalhar na loja número 20 vai definir a sua Loja coluna nulo. Em seguida, a fila para a Loja número 20 no quadro da loja será excluído;
 - **Em cascata.** Significa que as colunas na tabela de referencia serão também apagadas. No exemplo, todas as linhas na tabela FUNCIONARIO que trabalha na loja número 20 será suprimido. Em seguida, a fila para a Loja número 20 no quadro da loja será excluído.

3.4. Desencadeamento de Operações

A operação está acionando uma afirmação ou regra que rege a validade dos dados de manipulação operacionais tais como inserir, atualizar e excluir. Os componentes para desencadear uma operação são os seguintes:

- Regra de uso é uma declaração concisa do negócio regra a ser executada pelo desencadeamento operação;
- Evento que é a manipulação operação dados (inserir, atualizar ou excluir) que inicia a

operação;

- Nome da entidade que é o nome da entidade a ser acessadas ou modificadas;
- Condição que faz a operação a ser desencadeada;
- Ação que deve ser tomada quando a operação é desencadeada.

Figura 10 mostra um exemplo de como desencadear uma operação baseada na Montante em RETIRADA. Neste exemplo, se o pedido se tenta inserir uma linha na tabela a retirada no caso de o valor é mais do que o saldo da conta, o sistema irá rejeitar a operação. Não se pode retirar mais dinheiro do que aquilo que nos é encontrado em uma conta do.

```
Regra de Uso: WITHDRAWAL AMOUNT não deverá exceder ACCOUNT BALANCE
Evento: Insert
Nome da Entidade: WITHDRAWAL
Condição: WITHDRAWAL AMOUNT > ACCOUNT BALANCE
Ação: Rejeitar a insercao de transacoes
```

Figura 10: O exemplo desencadeia a operação

4. Volume de dados e análise de uso

Uso de Dados e Análise Volume fazem parte do banco de dados físicos desenho processo. Ao realizar análises volume dados, as estimativas do tamanho do banco de dados são utilizados para selecionar física dispositivos de armazenamento e estimar o custo de armazenamento. Ao realizar análise de utilização, as estimativas de uso caminhos ou padrões são utilizados para selecionar organizações arquivo e de acesso aos métodos plano para o uso de índices e de planejar uma estratégia para a distribuição dos dados. Uma maneira fácil de mostrar as estatísticas sobre os volumes e os dados de utilização é a notação, acrescentando que a entidade-que representa a relação diagrama final conjunto de relações normalizadas a partir de dados desenho lógico.

4.1. Análise do volume de dados

Considere o diagrama entidade-relacionamento das relações da Clínica de Repouso San Nicolas Baranggay na figura abaixo.

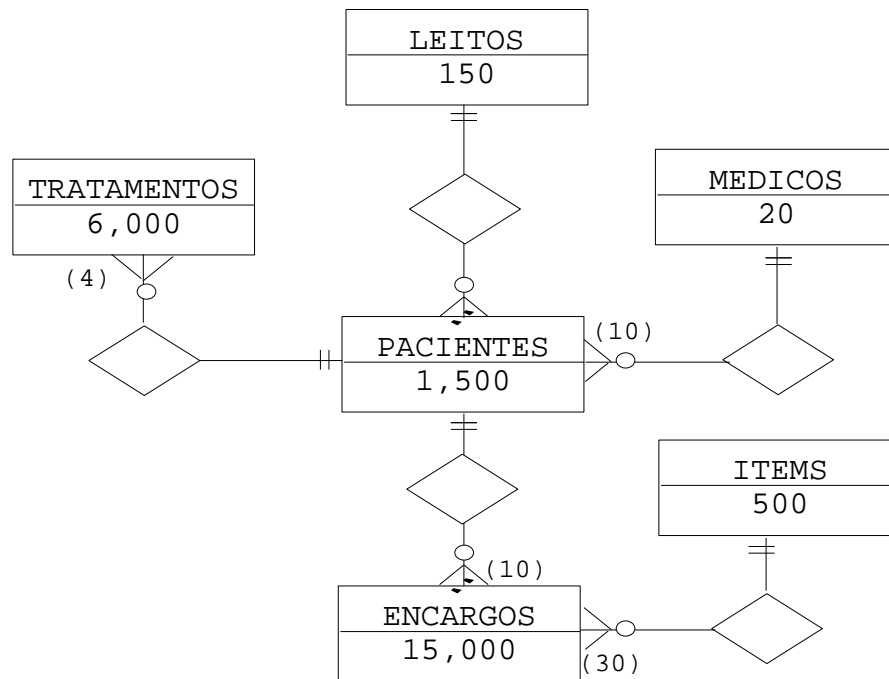


Figura 11: Clínica de Repouso San Nicolas Baranggay

Os dados relativos ao volume de registros ou filas são recolhidos, normalmente, durante a fase dos requisitos de engenharia ou na fase de análise. Números no interior das entidades serão a média do número de registros apresentados em determinado momento, relativamente. O número dentro do parênteses é o número de registro associado ao relacionamento um-para-muitos. A descrição da análise do volume de dados é a seguinte:

1. Desde que existam 150 leitos no hospital, o número máximo de pacientes admitidos em qualquer circunstância é limitado a 150
2. Em média, o registro do paciente está ativo por cerca de 30 dias
3. Em média, a duração da estadia do paciente é de 3 dias
4. O número total de pacientes ativos é de aproximadamente 1500 registros. Ele é computado como $x \text{ 150 leitos (30 dias para os prontuários ativos / 3 dias de duração da estada)}$
5. Após o período de 30 dias, o registro do paciente é arquivado
6. A média de paciente incorre numa média de 10 cobranças durante a hospitalização. A média do número de entidades COBRANCA espera-se que seja $(10 \text{ taxas} \times 1500 \text{ prontuários})$ ou 15000 registros de cobranças
7. Há 500 itens distintos que podem ser exibidos na fatura de um paciente. O número médio de cobranças pendentes para cada fatura é de $(15000 \text{ cobranças} / 500)$ 30 itens
8. Cada paciente recebe uma média de 4 tratamentos. O número médio de entidades TRATAMENTOS no banco de dados é $(4 \text{ tratamentos} \times 1500 \text{ doentes})$ 6000
9. Um médico examina uma média de 10 pacientes por mês

4.2. Uso da análise de dados

O analista identifica as principais transações e processamentos solicitados ao banco de dados. Cada operação de processos é então analisada para determinar os caminhos de acesso utilizados e estimar a frequência de uso. Faça uso do transaction map (mapa de transações). Quando todas as transações foram analisadas, o composite load map é elaborado, mostrando o total de caminhos de acesso usados no modelo conceitual.

O transaction map é um diagrama que mostra a sequência lógica de acesso de dados. A Figura 5 mostra o transaction map da Clínica de Repouso San Nicolau Baranggay sempre que a fatura de um doente é criada.

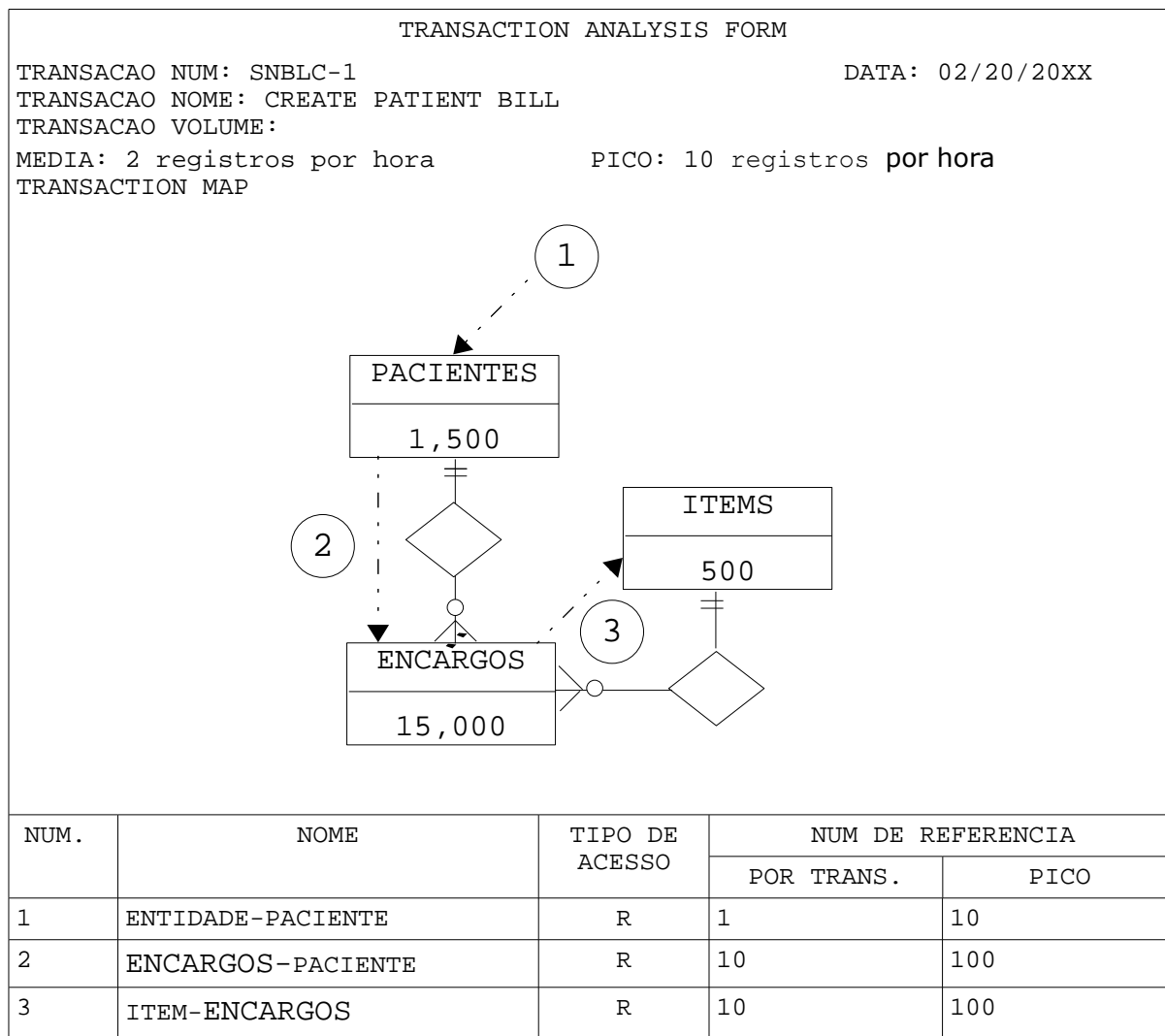


Figura 12: Criar Transaction Map do paciente Bill

1. A operação CREATE PATIENT BILL faz o registro do PACIENTE que será lido juntamente com os detalhes das despesas do paciente e também promove a impressão da fatura;
2. O volume médio de transações é de 2 por hora e 10 por hora durante o período de pico;
3. A operação exige apenas um PACIENTE como referência por transação, quando seu período de pico for 10 transações por hora;
4. Existem 10 ENCARGOS por PACIENTE. A média do número de vezes que o caminho ENCARGOS-PACIENTE é usado por transação é 10. Durante o período de pico, será 10 x 10 ou 100 por hora;
5. Uma vez que ITEM-ENCARGOS é atravessado uma vez por cada ENCARGOS, os dados de utilização também atingem um pico de utilização de 100 por hora.

O **Composite Load Map** é uma referência confiável para estimar o volume e o uso de dados no banco de dados.

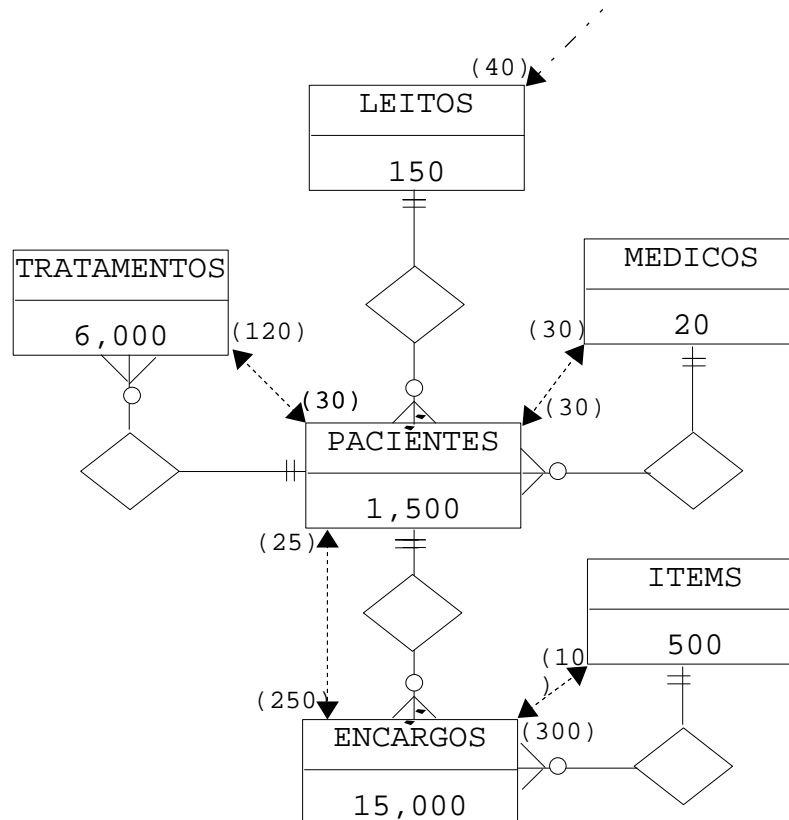


Figura 13: Composite Load Map da Clínica de Repouso San Nicolau Baranggay

1. O número à frente de cada seta tracejada é uma estimativa do número total de referências de acesso a um determinado caminho durante o pico de acessos;
2. Há registros de 40 leitos em período de pico;
3. O número de referências para a entidade a partir da entidade paciente está estimado em 120 por hora;
4. O número de referências ao paciente a partir do exterior do modelo é de 40 por hora;
5. O número de referência para os encargos da entidade o paciente entidade é estimada em 250 registros por hora;
6. O número de referências para os encargos da entidade PONTOS entidade é de 300 por hora.

Neste exemplo, nós achamos que o mais referências a entidade está ENCARGOS entidade.

5. Organização de Arquivos e Método de Acesso

Esta seção lida com os conceitos no que respeita ao armazenamento físico da base de dados secundários em dispositivos de armazenamento como discos rígidos e discos ópticos. A memória principal do computador é insuficiente para armazenar o banco de dados. Embora, por vezes o acesso primário de armazenamento são muito mais rapidamente do que armazenamento secundário, não é suficientemente grande para armazenar a quantidade de dados que pode exigir um típico banco de dados. Há alguns constrangimentos que precisa ser considerado na escolha do tipo de armazenamento secundário. São eles:

- características físicas do armazenamento secundário
- sistema operacional disponível
- software de gestão de arquivos
- necessidades do usuário para armazenamento e acesso aos dados

O banco de dados de armazenamento secundário é organizado em um ou mais arquivos, onde

cada arquivo é constituído por um ou mais registros. Cada registro terá, então, um ou mais campos. No dia anterior, um arquivo corresponde a uma tabela, um registro corresponde a uma linha e uma campo corresponde a uma coluna.

O registro físico é a unidade de transferência entre disco e memória, e vice-versa. Também é conhecida como bloco ou página, eles consistem em mais do que apenas um registro lógico. Embora, por vezes, dependendo do tamanho, um registro lógico pode corresponder a um registro físico. Hoje em dia, um banco de dados corresponde a um ou mais arquivos que podem ocupar discos inteiros; discos são divididos em páginas ou blocos.

A ordem na qual os registros são armazenados e acessos aos arquivos dependem de sua organização. **File Organization** (Organização de Arquivos) é uma técnica para arrumar fisicamente os registros de um arquivo em um dispositivo de armazenamento secundário. É a forma física dos dados em um arquivo de registros e páginas de armazenamento secundário. Geralmente, há duas organizações de arquivo. Sendo as seguintes:

- Sequential File Organization (organização seqüencial de arquivos)
- Hashed File Organization (organização aleatória de arquivos)

Junto com a organização de arquivos, existe um conjunto de métodos para acesso aos arquivos. O método de acesso aos arquivos define as etapas envolvidas ao armazenar e recuperar registros a partir de um arquivo. Em geral, há três métodos de acesso aos arquivos. Sendo:

- método de acesso seqüencial
- método de acesso indexado
- método de acesso aleatório ou direto

Alguns métodos de acesso podem ser aplicados apenas em determinadas organizações de arquivos. Por exemplo, o método de acesso aleatório não pode ser aplicado na organização seqüencial de arquivos.

Veremos a seguir os critérios para selecionar uma organização de arquivos:

1. rapidez na recuperação
2. agilidade no processamento de transações
3. utilização eficiente de espaço de armazenamento
4. proteção para falhas ou perda de dados
5. diminuir a necessidade de reorganização
6. possibilitar o crescimento
7. segurança, para evitar uso não autorizado

5.1. Organização Seqüencial de Arquivos

A organização de arquivos seqüencial é o tipo mais comum de organização de arquivos. Os registros são armazenados de forma a registrar o campo-chave. Outro nome de arquivo seqüencial é pilha. Como exemplo, considere que a Figura 7 mostra a forma como os registros da tabela LOJA são armazenados em disco. Elas são organizadas de acordo com seu número, onde o menor número da loja é o primeiro da lista. Os métodos de acesso aos arquivos, que podem ser usados para recuperar registros em um arquivo seqüencial, são os seguintes:

Método de Acesso Seqüencial

Para localizar um determinado registro, o usuário normalmente procura o arquivo a partir do início até onde o registro está localizado. Isto, torna a recuperações de registros relativamente lenta. Para inserir ou atualizar um registro, é necessário ir até o arquivo. Porque os registros não podem ser inseridos no meio do arquivo, um arquivo seqüencial é normalmente copiado durante a atualização em processo.

Para eliminar um registro, sua primeira página deve ser recuperada, o registro é marcado como apagado. Quando a página é escrita de volta no disco, seu espaço é automaticamente excluído dos registros e reutilizado. Os arquivos são periodicamente reorganizados pelo administrador do banco de dados, para recuperar o espaço disponibilizado por registros suprimidos.

Método Indexado de Acesso aos Arquivos

Discutido na seção de índices desta lição.

LOJA

Numero	Nome	Endereço
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	NorthDakota
40	GangStore in Michigan	Detroit, Michigan
50	GangStore in South Carolina	South Carolina

Figura 14: Exemplo de arquivo para armazenar a tabela seqüencial

5.2. Hashed File Organization

Outra organização de arquivos é a organização de arquivo *hash*, onde os registros são armazenados no disco aleatoriamente. O endereço para cada registro é determinada usando um algoritmo *hash*. Algoritmo *hash* ou função *hash* é uma rotina que calcula o endereço do registro. O campo *hash* é o domínio no qual a base calcula o endereço. Normalmente, esta é a principal chave do registro. Neste caso, a área *hash* é conhecida como a chave *hash*. Registros dentro de um arquivo *hash* aparecem aleatoriamente distribuídos em todo o espaço disponível em disco. Por esta razão, os arquivos *hash* são muitas vezes chamado aleatórios ou arquivos diretos.

Um método comum utilizado para o algoritmo ou função *hash* é o restante da divisão. Dividir o número original por um número primo que se aproxima da quantidade a ser armazenada. Em seguida, o resto da divisão é usado como o endereço. Por exemplo, suponha que uma empresa tem 1000 funcionários registrados. Uma boa aproximação de valor seria 997, porque é perto de 1000. Pense em um registro tendo a chave primária 50542. 50542 dividido por 997. O restante é 629. Esta é a localização (endereço da página ou bloco) do registro no disco.

Algoritmos ou funções *hash* não garantem um endereço exclusivo, porque o número de possíveis valores *hash* que um campo pode ter, geralmente, é muito maior do que o número de endereços disponíveis para os registros.

Cada endereço gerado a partir de uma função *hash* corresponde a uma página com saídas para vários registros. Dentro da página, os registros são inseridos em sua ordem de chegada. Quando o mesmo endereço é gerado por dois ou mais campos *hash* diferentes, dizemos que ocorreu um conflito. Quando isso acontece, o novo registro deve ser inserido em outro local. Gerenciar conflitos complica a gestão de arquivos *hash* e degrada o desempenho global. Existem várias técnicas que podem ser usadas para gerenciar conflitos:

1. Gerando endereços. Quando há um conflito, o registro é inserido no primeiro endereço disponível. Quando a última página foi pesquisada e não existem faixas disponíveis, o sistema inicia uma nova página. Consideremos o exemplo na figura 8, onde uma página pode conter um máximo dois registros (duas faixas disponíveis). Suponhamos que temos de inserir o registro para a Loja 50 na tabela LOJA. Suponhamos também, que quando o algoritmo *hash* foi utilizado, ele enviou para o endereço da página 1. Note que não há mais faixas disponível. O sistema irá pesquisar linearmente a primeira faixa que estiver disponível na página 2 faixa 1;

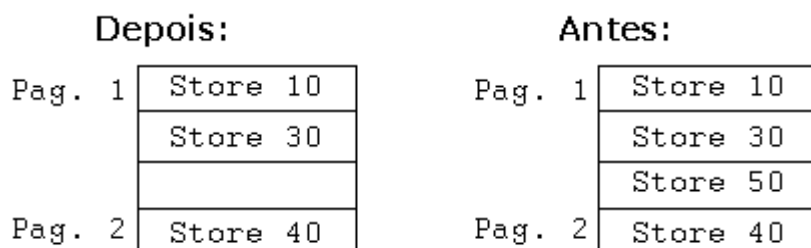


Figura 15: Gerando Endereços

2. Estouro de memória desencadeado. Quando há um colisão, um transbordamento é mantida área que não pode ser colocada no endereço *hash*. O registro é inserido na área

de overflow. Considere o exemplo mostrado na Figura 9. Suponhamos que temos de inserir o registro para a Loja 50 na tabela LOJA. Suponhamos também, que quando o algoritmo *hash* é usado, ele levou para o endereço da página 1. Note que não há mais slot disponível. O sistema irá colocar o registro no overflow. Neste exemplo, é Pag. 2 faixa 1.

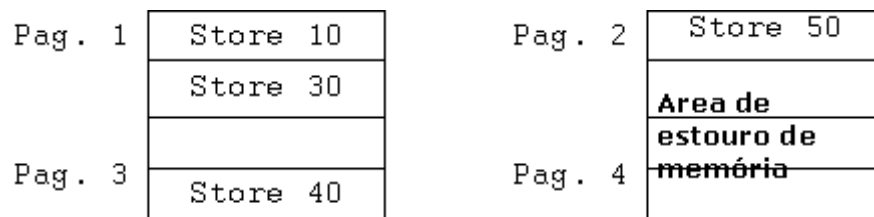


Figura 16: Estouro de memória

3. Estouro de memória encadeado. Esta é uma variação do estouro de memória desencadeado. Uma área de estouro de memória é mantida para serem colocados os endereço *hash*. Uma outra área é usada pela página. O campo, também conhecido como sinônimo de ponteiro e indica se um conflito ocorreu. Pag. 1 aponta para a área que está sobrando Pag. 2. Pag. 3 não tem nenhum espaço sobrando.

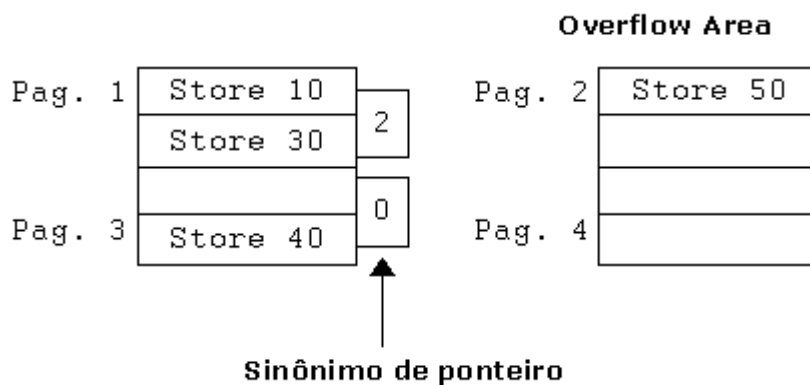


Figura 17: Estouro de memória

4. Múltiplos hashes. Quando um conflito ocorre, uma segunda função hash é utilizada. O objetivo é produzir um novo endereço hash que permitirá evitar o conflito. A segunda função hash é usada para fazer registros em uma área de estouro de memória.

6. Índices

Nesta seção, serão discutidas técnicas para fazer o retorno de dados mais eficiente usando índices. Um índice é uma tabela ou outra estrutura de dado que é usada para determinar a localização de linhas em uma tabela (ou tabelas) que satisfaçam alguma condição. Pode ser definido como chave primária e valores de atributo sem chave. São mais compactos que os registros de dados (ou linhas) a que fazem referência. Podem ser alocados em memória durante períodos longos a fim de reduzir o custo de acesso a memória secundária para retorná-los.

Uma estrutura de índice é associada com uma chave especial de procura, e contém registros que consistem do valor da chave e o endereço do registro lógico no arquivo que contém o valor da chave. O arquivo de dado é um arquivo contendo os registros lógicos enquanto que o arquivo de índice contém os registros de índice. Os registros no arquivo de índice são ordenados de acordo com o campo de indexação, que é geralmente baseado em uma coluna.

Quando usamos índices?

- Devem ser generosamente para aplicações que são usadas primeiramente para suportar retorno de dados como as aplicações de suporte a decisão.
- A contra-partida entre a performance melhorada através do uso de índices e a queda de performance pelos métodos de inserção, deleção, e atualização de registros deve ser compreendida. A performance cai quando se modifica o conteúdo da tabela porque também se precisa atualizar os índices das tabelas.

6.1. Arquivos Seqüencialmente Indexados

Arquivos de dados armazenam registros seqüencialmente ou de forma não-seqüencial, e um índice é criado para permitir que as aplicações localizem registros individualmente. Existem dois tipos básicos de organização de arquivos por índice. São eles:

Organização de arquivos com índice em seqüência: Os registros são armazenados seqüencialmente pelo valor da chave primária. Um índice simples, chamado índice de bloco ou índice primário, pode ser usado. Esta estrutura é um compromisso entre um arquivo puramente seqüencial e um arquivo puramente aleatório, podendo ser processada seqüencialmente ou individualmente acessada usando uma chave de procura que acessa o registro pelo índice. Uma organização de arquivo com índice em seqüência, normalmente, tem:

- uma chave primária
- um índice ou índices separados
- uma área de overflow (estouro)

Um método largamente usado é o Método de Acesso Seqüencial-Indexado (ISAM) definido pela IBM. Um índice é dito esparsos se somente alguns valores da chave de busca tem índices. Um índice é dito denso se cada um dos valores da chave busca tem índices. A seguinte figura ilustra isso:

Dense Index**Index:**

Store 10
Store 30
Store 40
Store 50

Data File:

Store 10
Store 30
Store 40
Store 50

Page 1

Page 2

Sparse Index**Index:**

Store 10
Store 40
Store 60

Data File:

Store 10
Store 30
Store 40
Store 50

Page 1

Page 2

Figure 18: Dense vs. Sparse Index

Um exemplo de organização de arquivos com índice em sequência, é uma lista de telefones. No canto esquerdo da página, pode-se achar o último nome da primeira pessoa daquela página. No canto direito, pode-se achar o último nome da última pessoa daquela página. Outro exemplo é ilustrado na figura 19. Aqui, os registros da tabela **STORE** (arquivo de dados) são guardados em forma sequencial de acordo com a chave primária NUMBER. O **STORE-ADDRESS-INDEX** (**arquivo índice**) contém os endereços organizados em ordem alfabética, e cada endereço aponta para o registro na tabela **STORE**.

STORE-ADDRESS-INDEX

Alabama	1
Detroit, Michigan	4
North Dakota	3
South Carolina	5
West Virginia	4

STORE

	Number	Name	Address
1	10	GangStore in Alabama	Alabama
2	20	GangStore in West Virginia	West Virginia
3	30	GangStore in North Dakota	NorthDakota
4	40	GangStore in Michigan	Detroit, Michigan
5	50	GangStore in South Carolina	South Carolina

Figure 19: Exemplo Indexed File Organization

Organização de Arquivos com Índices Não-Sequenciais: Os registros são guardados não-sequencialmente, um índice completo, chamado índice inverso, é necessário. Um exemplo são os livros em uma livraria, onde são guardados de acordo com o número do catálogo, e não de acordo com o nome do autor e título. Os catálogos de título e autor são índices completos que permitem uma pessoa encontrar rapidamente a posição correta de um certo livro. Outro exemplo é mostrado na figura a seguir, os dados na tabela **STORE** não são armazenados de acordo com a chave primária **NUMBER**. O **STORE-ADDRESS-INDEX** (índice do arquivo) contém o endereço dos registros organizados em ordem alfabética, e cada endereço aponta para o registro na tabela **STORE**.

STORE-ADDRESS-INDEX

Alabama	4
Detroit, Michigan	3
North Dakota	1
South Carolina	5
West Virginia	2

STORE

	Number	Name	Address
1	30	GangStore in North Dakota	North Dakota
2	20	GangStore in West Virginia	West Virginia
3	40	GangStore in Michigan	Detroit, Michigan
4	10	GangStore in Alabama	Alabama
5	50	GangStore in South Carolina	South Carolina

Figure 20: Indexed Non-sequential File Organization

6.2. Índices em Cluster

Se o campo indexado não é um campo chave da tabela, então pode ser mais que um registro correspondendo ao mesmo valor no campo indexado, o índice é chamado de índice em cluster ou índice secundário. São índices baseados em atributos não-chave. Normalmente, são usados quando registros em um arquivo são retornados frequentemente baseados nestes valores. O propósito é aumentar a velocidade das respostas retornadas ordenando fisicamente os registros do índice do arquivo naquele atributo não-chave, chamado atributo em cluster, que é usado para juntar ou agrupar linhas que tem um valor em comum para esse atributo. Há várias técnicas para lidar com índices secundários não-únicos.

1. Criar um índice secundário grande que mapeia para todos os registros no arquivo de dados permitindo que valores duplicados de chaves apareçam no índice
2. Ter um índice secundário para ter uma entrada no índice para cada valor de chave distinto e ter os apontadores de bloco sendo multi-valorados com uma entrada correspondente a cada valor duplicado de chave no arquivo de dados
3. Ter um índice secundário para ter uma entrada no índice para cada valor de chave distinto mas o ponteiro de bloco não apontaria para o arquivo de dados e sim para uma página que contém ponteiros para os registros correspondentes no arquivo de dados

A próxima figura mostra um exemplo de um arquivo de índice baseado em atributos não-chave. O DESCRIPTION-INDEX é um arquivo de índice para a coluna DESCRIPTION da tabela PRODUCT. Quando há o retorno de registros da tabela PRODUCT baseado na descrição do produto tal que a busca por todos os produtos com a descrição de DESK, o arquivo de índice DESCRIPTION-INDEX é primeiro procurado por todos os valores que são iguais aos da busca. Veja, que o arquivo de índice mostra que os registros 5 e 9 são produtos tendo DESK como descrição.

DESCRIPTION-INDEX

DESCRIPTION	RECORD NO.
Chair	4, 6
Book Shelf	8
Desk	5, 9
Stand	7

PRODUCT

	Number	Description	...
4	1000	Chair	
5	3500	Desk	
6	6250	Chair	...
7	9750	Stand	
8	1250	Book Shelf	...
9	1425	Desk	

Figura 21: Índices para os atributos não-chaves

6.3. Índices de Multi-Nível

Quando um arquivo de índice torna-se grande e se estende para várias páginas, o tempo necessário para buscar os registros aumenta. Um índice multi-nível tenta reduzir o alcance das buscas. Ele faz isso tratando o índice como um arquivo qualquer, divide o índice em outros índices menores e mantém um índice para esses outros índices. A figura 21 mostra um exemplo. Cada página no arquivo de dados armazena dois registros; na verdade, uma página compõe mais registros. Cada registro de índice armazena um valor de chave de acesso e um endereço de página. O valor chave armazenado é o maior na página endereçada.

Para localizar um registro, se inicia pelo índice de primeiro nível e procura-se na página pelo último acesso ao valor da chave que é menor ou igual à chave do registro. Por exemplo, procuramos a STORE 39, pelo índice de primeiro nível, nós achamos uma chave que é menor ou igual a este valor, nesse caso, STORE 37. Esse registro contém o endereço para o índice de segundo nível da página. Para continuar a busca, nós achamos uma chave que é menor ou igual a chave dos registros. Nesse caso, é a STORE 37 que contém o endereço da página onde a STORE 39 está localizada. Repetindo o processo chegamos a terceira página (terceiro nível).

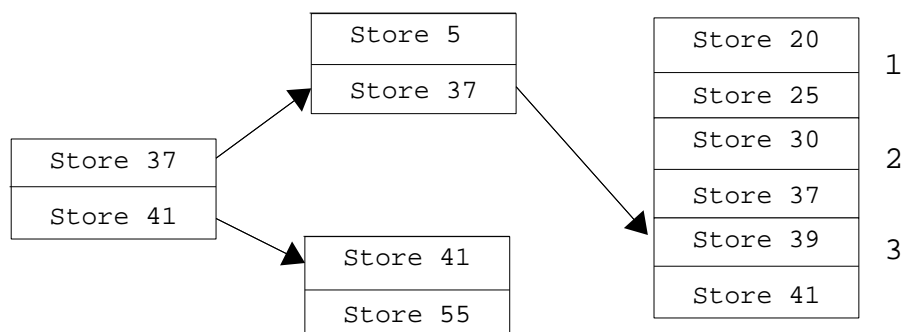


Figura 22: Multi-level Indexes

6.4. Índices Árvores

Uma árvore é uma estrutura de dados que consiste de um grupo de nós que saem de um outro nó. Isso forma uma árvore de cabeça para baixo. Consiste do seguinte:

1. **Nó raiz:** é o nó no topo da árvore
2. **Nó folha:** é um nó em uma árvore que contém um nó filho
3. **Subárvore:** é um nó e todos os descendentes daquele nó
4. **Ponteiro:** é um campo contendo dados que podem ser usados para achar um registro relacionado

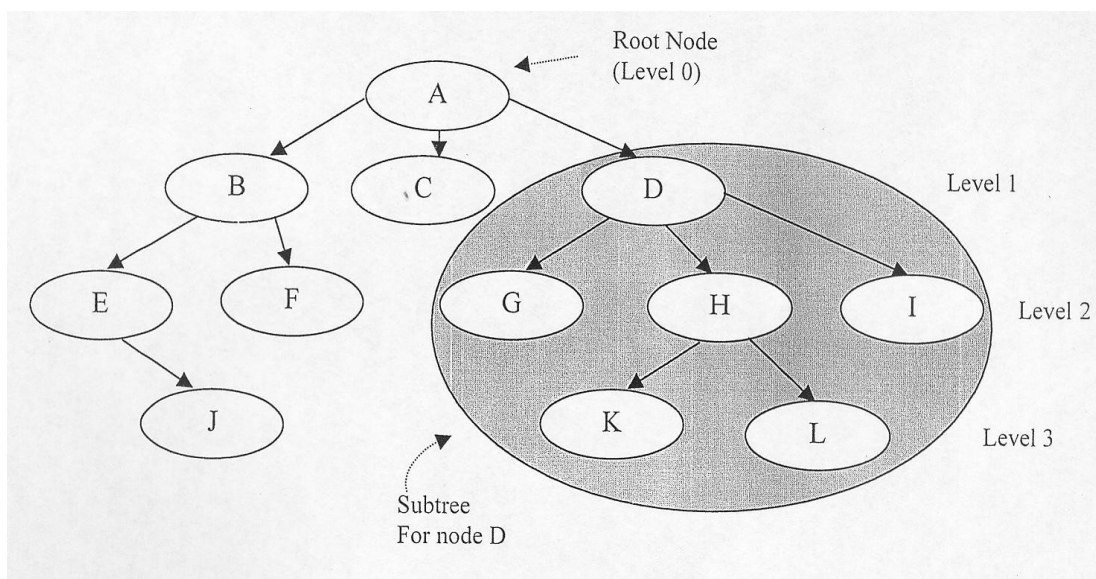


Figura 23: Estrutura de Árvore

Uma árvore tem três propriedades a seguir:

1. **Acessibilidade Uniforme:** é alcançada quando todos os nós folha têm a mesma distância da raiz. Os registros acessados frequentemente são colocados perto do nó raiz para se ter menos acesso ao disco. O acesso ao disco ocorre quando se passa entre níveis.
2. **Fator de bifurcação:** é também conhecido como o grau da árvore. É o número máximo de nós filhos permitido por um nó pai. Uma grande bifurcação geral, em geral, mais árvores superficiais.
3. **Profundidade:** é o número de níveis entre o nó raiz e um nó folha na árvore. Quando a profundidade é a mesma do nó raiz para cada nó folha, a árvore é dita balanceada.

Árvores B+ são árvores onde todas as folhas estão a uma mesma distância da raiz. É uma árvore binária de ordem 2 onde cada nó não tem mais do que dois filhos. As regras para uma árvore B+ são as seguintes:

1. Se a raiz não é um nó folha, deve ter ao menos dois filhos.
2. Para uma árvore de ordem n , cada nó, exceto a raiz e os nós folha, deve ter entre $n/2$ e n ponteiros e filhos. Se $n/2$ não for inteiro, é arredondado.
3. Para uma árvore de ordem n , o número de valores chave em um nó folha deve ser entre $(n-1)/2$ e $(n-1)$ ponteiros e filhos. Se $(n-1)/2$ não for inteiro, o resultado é arredondado.
4. O número de valores chave contidos em um nó não-folha é 1 número menor do que o número de ponteiros.
5. A árvore deve sempre estar balanceada, ou seja, cada caminho do nó raiz para um nó folha deve ter o mesmo tamanho.
6. Nós folha são ligados em ordem de valores de chave.

A figura 23 mostra um exemplo de uma árvore B+.

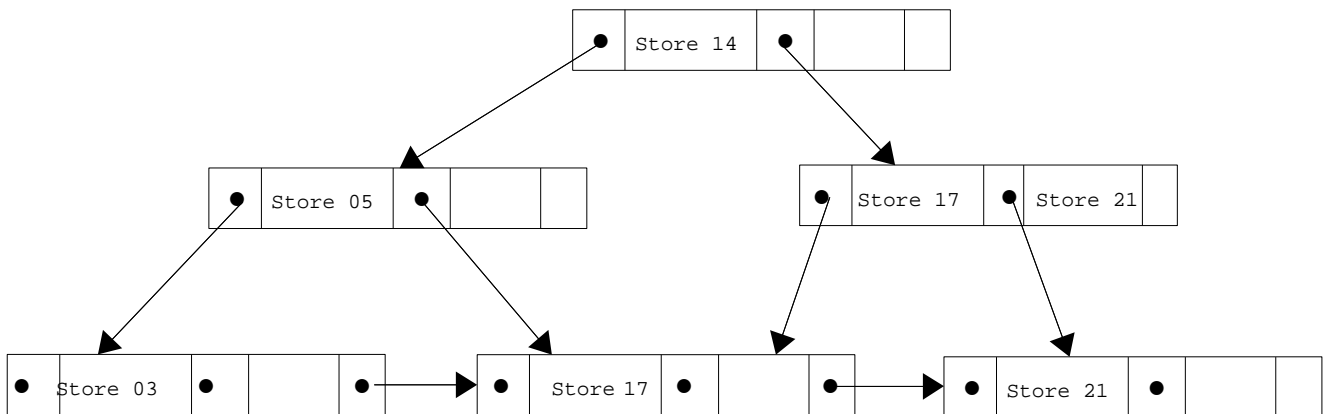
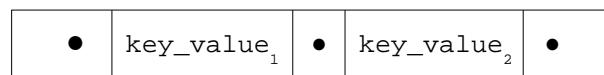


Figura 24: B+Tree Structure

Cada nó é da forma abaixo:



Onde • representa um ponteiro para outro registro ou vazio (ponteiro para lugar nenhum). Se o valor da chave de busca é menor ou igual a key_value_i , o ponteiro a esquerda de $[]$ é usado para achar o próximo nó a ser procurado. Senão, o ponteiro para o fim do nó é usado. Por exemplo, queremos achar o registro chamado STORE 21. A busca começa do nó raiz, STORE 21 é maior que STORE 14. Seguimos o ponteiro para a direita, o que nos leva ao nó de segundo nível contendo os valores STORE 17 e STORE 21. Continuamos seguindo o ponteiro pela esquerda o que nos leva ao registro de STORE 21.

6.5. Desnormalização

Desnormalização é o processo de transformar tabelas normalizadas em especificações de registros

físicos anormalizados. Em geral, a desnormalização deve:

- Combinar colunas de várias tabelas para formar um registro físico.
- Particionar uma tabela em vários registros físicos.
- Fazer uma combinação das duas opções acima.

Um cuidado deve ser tomado quando for desnormalizar tabelas porque aumenta a chance de erros ou inconsistências. Às vezes, pode até forçar a reprogramação de aplicações devido a mudança das regras de negócio. Pode otimizar certos processamentos de dados ao custo de outras operações.

Abaixo uma lista de situações em que comumente a desnormalização é considerada:

1. **Duas entidades com um relacionamento um-para-um.** Mesmo se uma das entidades for um participante opcional, se a entidade que é procurada existe a maior parte do tempo, então talvez seja inteligente combinar estas duas entidades em uma tabela.
2. **Um relacionamento muitos-para-muitos com atributos não-chave.** Ao invés de causar a junção de tabelas de árvore para extrair dados das duas entidades no relacionamento, avisará para combinar colunas de uma das entidades dentro da tabela representando o relacionamento muitos-para-muitos e evitando uma junção em muitas pesquisas.
3. **Referência a dados.** Dados referenciados existem em uma entidade do lado-um num relacionamento um-para-muitos, e essa entidade não participa em nenhum outro relacionamento de base de dados.

A maioria das oportunidades mostradas acima concordam com a combinação de tabelas para minimizar a operação de junção. A desnormalização pode ser usada para criar mais divisão ou particionamento de tabelas em muitas outras tabelas. O particionamento pode ser:

- **Particionamento Horizontal:** quebra uma tabela em múltiplas especificações de registros colocando linhas diferentes em diferentes tabelas baseado em valores comuns. Considere o inventário para cada loja em THE ORGANIC SHOP. A tabela INVENTORY pode ser dividida baseando-se no valor da coluna STORE_NO desde que o processamento do inventário de uma loja é feito separadamente de outra loja. Veja a figura 24 para a ilustração de um particionamento horizontal. Observamos que cada tabela teria a mesma estrutura. Cada linha dentro de cada tabela são registros para uma loja específica, ou seja, INVENTORY_10 conterá apenas registros para STORE No. 10, INVENTORY_20 só conterá registros para STORE No 20, etc, INVENTORY_10 e INVENTORY_20 são chamados de uma partição da tabela INVENTORY.

INVENTORY			
Store_No	Item_Code	Count	Operational Level
10	1001	50	20
20	1001	2300	500
10	1004	4500	100
20	1004	90	100

AFTER HORIZONTAL PARTITIONING			
INVENTORY_10			
Store_No	Item_Code	Count	Operational Level
10	1001	50	20
10	1004	4500	100

INVENTORY_20			
Store_No	Item_Code	Count	Operational Level
20	1001	2300	500
20	1004	90	100

Figura 25: Particionamento Horizontal

Alguns detalhes importantes sobre o Particionamento Horizontal:

1. Faz sentido quando categorias diferentes de linhas de uma tabela são processadas separadamente. No exemplo, há o processamento do inventario da loja numero 10 separadamente do inventario da loja 20.
 2. Pode ser seguro porque podemos limitar o acesso de usuários a certas linhas de dados. Empregados da loja numero 10 so podem acessar INVENTORY_10; não podendo acessar INVENTORY_20.
 3. Pode permitir deixar uma partição 'sem serviço' porque ela foi danificada ou então ela pode ser recuperada permitindo ainda o processamento de outras partições enquanto o problema persistir.
 4. Pode permitir partições diferentes residirem em discos físicos diferentes.
- **Particionamento Vertical:** é a distribuição das colunas de uma tabela em muitas tabelas separadas. Como um exemplo, supondo queremos que apenas certas aplicações possam acessar e modificar a coluna "OPERATIONAL LEVEL"; outras aplicações acessam e modificam a coluna "COUNT". Isto é por propósitos de segurança. Podemos dividir a tabela INVENTORY como mostrado na Figura 25. A coluna INVENTORY_COUNT pode ser acessada por aplicações que gerenciam reservas e vendas de estoques enquanto a INVENTORY_LEVEL pode ser acessada por aplicações que gerenciam compras e manutenções. INVENTORY_COUNT e INVENTORY_LEVEL são as partições.

INVENTORY			
Store_No	Item_Code	Count	Operational Level
10	1001	50	20
20	1001	2300	500
10	1004	4500	100
20	1004	90	100

AFTER VERTICAL PARTITIONING			
INVENTORY_COUNT			
Store_No	Item_Code	Count	
10	1001	50	
10	1004	4500	

INVENTORY_LEVEL			
Store_No	Item_Code	Level	
20	1001	500	
20	1004	100	

Figura 26: Particionamento Vertical

Combinações de particionamento vertical e horizontal são possíveis. Essa forma de desnormalização é comum para uma base de dados que é distribuída entre muitos computadores. A discussão de base de dados distribuída está além do escopo desse curso.

Uma outra forma de desnormalização é a replicação de dados onde os mesmos dados estão propositalmente armazenados em lugares múltiplos. A discussão a respeito de replicação de dados está fora do escopo desse curso.

7. Exercícios

Desenvolver o modelo físico da base de dados a partir do modelo lógico desenvolvido do problema 3 na lição 3 (o gerenciador de apartamento). Defina o seguinte:

- Restrições de domínio
- Integridade da entidade
- Integridade referencial
- Operações com gatilho

Desenvolver o modelo físico da base de dados gerada do problema 4 na lição 3 (o gerenciador de loja de eletrônico).

- Restrições de domínio
- Integridade da entidade
- Integridade referencial
- Operações com gatilho

Desenvolver o modelo físico da base de dados desenvolvida para o sistema de distribuição e-Lagyan.

- Restrições de domínio
- Integridade da entidade
- Integridade referencial
- Operações com gatilho

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.

Banco do Brasil

Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec

Suporte e apoio financeiro e logístico a todo o processo.

Borland

Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB

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

Módulo 9

Banco de Dados



Lição 5

Structure Query Language (SQL)

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

1. Objetivos

Nesta lição discutiremos a SQL – Structured Query Language (Linguagem Estruturada de Consulta), desde de que os sistemas de bancos de dados relacionais passaram a usar esta linguagem, os objetivos são:

- SQL forneça uma pequena instrução com fundamento matemático de linguagens relacionais e cálculos relacionais e álgebra relacional
- Aprender os dois componentes, chamados de DDL – *Data Definition Language* (Linguagem de Definição de Dados) e DML – *Data Manipulation Language* (Linguagem de Manipulação de Dados)
- Aprender como criar e remover gatilhos

Ao final desta lição, o estudante será capaz de:

- Entender o funcionamento de comandos de modificação em SQL
- Compreender o funcionamento de comandos de consulta em SQL

2. Linguagem Relacional

Uma das partes dos modelos relacionais é a linguagem que define que tipo de operação nos dados que é permitida. Isso inclui recuperação e alteração de dados da base de dados, e operações que modificam a estrutura da base de dados, existem dois tipos de linguagem usadas pelas bases de dados relacionais para manipulação de dados.

- **Linguagem de procedimentos** onde o usuário especifica exatamente como manipular os dados
- **Linguagem não procedimental** onde o usuário especifica que dado quer antes de informar como ele será recuperado

Nesta seção discutiremos o cálculo da álgebra relacional e a base das linguagens relacionais. A álgebra relacional, informalmente, pode ser pensada como uma linguagem relacional de alto nível. Ela é usada para transmitir ao sistema gerenciador da base de dados como construir uma nova relação com uma ou mais relações na base de dados, por outro lado os cálculos relacionais podem ser pensados como uma linguagem não procedural que pode ser usada para formular definições de relações em termos de uma ou mais relações de banco de dados. Ambos são equivalentes a um outro, isto é, para muitas expressões em álgebra é o equivalente a expressões de cálculos e vice-versa

São usadas como a base para uma linguagem de manipulação de alto nível, tal como a DML das bases de dados relacionais. São de interesse porque elas ilustram as operações básicas precisadas por DML.

2.1. Álgebra Relacional

Álgebra relacional é uma linguagem teórica com operadores que trabalham com uma ou mais relações para definir outra relação sem alterar a relação original. Tanto os operandos quanto o resultado da operação são relações, isso faz com que a saída de uma operação seja a entrada de outra operação. Esta propriedade é conhecida como fechamento (closure), isto é, expressões podem ser aninhadas na álgebra relacional similarmente às operações aritméticas.

Álgebra relacional é conhecida como linguagem de relação em tempo real, isso significa que todas as tuplas possivelmente venham de diferentes relações, são manipuladas em uma declaração. Existem cinco operações fundamentais.

- **Seleção ou Restrição.** A operação de seleção trabalha em uma relação simples R e define a relação que contém apenas as tuplas (linhas) de R e satisfaz a condição (predicado).

$$\sigma_{\text{predicate}}(R)$$

Exemplo 1: Listar todos os cartões de telefone com o valor igual a 500.

$$\sigma_{\text{value} = 500}(\text{Cartão de telefone})$$

Neste exemplo, a relação de entrada é `PhoneCard` e o predicado é o valor igual a 500, a operação de seleção define a relação que contém unicamente a linha `PhoneCard` ou registros com valor igual a 500.

Predicados mais complexos podem ser obtidos usando operadores lógicos \wedge (AND), \vee (OR), and \sim (NOT).

- **Projeção.** A operação de projeção trabalha em uma relação simples R e define a relação que contém um subconjunto vertical de R, extraíndo valores de atributos especificados e eliminando valores duplicados.

$$\Pi_{\text{col1}, \dots, \text{coln}}(R)$$

Exemplo 2: Produz a lista de todos os cartões de telefone "PhoneCard" mostrando apenas o número do cartão provendo o id e o valor.

$$\Pi_{\text{cardno}, \text{supplierID}, \text{value}}(\text{PhoneCard})$$

Neste exemplo, a operação define a relação que contem apenas os atributos cardno, supplierID e value dos PhoneCards, nesta ordem especifica.

- **Produto cartesiano.** A operação produto cartesiano define a relação que a concatenação de todas as tuplas da relação R com todas as tuplas da relação S. Isso multiplica duas relações para definir outra consistindo em todos os pares de tuplas possíveis dentre duas relações. Significando dizer que em duas relações, onde uma possui M tuplas com I atributos, e outra relação possui N tuplas com J atributos, o resultado da relação conterá (M*N) tuplas com I*J atributos. É possível que a relação tenha o mesmo nome, neste caso os atributos serão prefixados pelo nome da relação de origem para manter a singularidade da definição dos atributos dos nomes.

$$R \times S$$

Exemplo 3: Listar todas as possíveis combinações de cardNo, supplierId, value, recipientID e dataLoaded de todas as contas.

$$(\Pi \text{ cardno, supplierID, value }^{(\text{PhoneCard})}) \times (\Pi \text{ recipient, dateLoaded }^{(\text{PhoneCardTrans})})$$

A informação relativa a cada cartão é dada para qual conta é armazenada em PhoneCardTrans. A informação sobre o provedor e o valor de cada cartão é encontrada na relação de PhoneCard.

A relação resultante conterá mais informação que o que é requerido. Muitas delas não reflete a correta combinação de tuplas. Para se obter a lista apropriada é preciso realizar uma operação de seleção de retirada destes registros PhoneCard.cardNo = PhoneCardTrans.cardNo.

$$\sigma_{\text{PhoneCard.cardNo} = \text{PhoneCardTrans.CardNO}} ((\Pi \text{ cardno, supplierID, value }^{(\text{PhoneCard})}) \times (\Pi \text{ recipient, dateLoaded }^{(\text{PhoneCardTrans})}))$$

- **União.** A união de duas relações R e S com M e N tuplas, respectivamente é obtida concatenando em uma relação onde o máximo de tuplas é (M+N), tuplas duplicadas são eliminadas. R e S tem que ser a união compatível i.e., é o esquema de duas partidas de relação. Elas tem o mesmo numero de atributos com dominios emparelhados.

$$R \cup S$$

Exemplo 4: Construa uma lista de todas as transações realizadas por cartões de telefone e cartões de acesso a internet.

$$\Pi \text{ cellPhoneNo, cardNo, dateLoaded, recipient }^{(\text{PhoneCardTrans})} \cup \Pi \text{ cellPhoneNo, internetCardNo, dateLoaded, recipient }^{(\text{InternetTrans})}$$

A operação de projeção é usada para pegar o cellPhoneNo, cardNo, dateLoaded,recipeiente da relação PhoneCardTrans. Então operação de projeção para pegar cellPhoneNo, internetCardNo, dateLoaded e o recipeiente de InternetTrans.

- **Diferença Fixa.** A operação "diferença fixa" define a uma relação consistindo em tuplas que estão na relação R mas não estão na relação S, R e S tem que ser uma união compatível.

$$R - S$$

Exemplo 5: construir uma lista de todos os cartões de telefone que não tem sido usados aplicando a operação "diferença fixa".

$$\Pi \text{ cardNo}^{(\text{PhoneCard})} - \Pi \text{ cardNo}^{(\text{PhoneCardTrans})}$$

Aqui a união de duas relações compatíveis são definidas. Uma é a execução de uma operação de projeção em PhoneCard que pega os cardNo's armazenados atualmente no banco de dados; a segunda é uma execução da operação de projeção em PhoneCardTrans pegando os cardNo's que tem sido carregados. As tuplas que estão em PhoneCard que não foram achadas na fixação das tuplas de PhoneCardTrans serão as tuplas de resultado.

Operações adicionais podem existir combinando as cinco relações fundamentais da álgebra relacional.

- **Unir.** A operação de junção "join" é uma das operações fundamentais da álgebra relacional. Ela combina duas relações para formar uma nova relação. Ela é derivada da operação de produto cartesiano o que é equivalente a execução do predicado da junção sobre o produto cartesiano. Existem varias formas de uma operação de junção "Join":

Theta-join (θ -join)

A operação teta-join define a relação que contem tuplas que satisfaçam o predicado F sobre o projeto cartesiano de R e S. O predicado de F é uma forma de $R.a_i \theta S.b_i$, onde θ pode ser uma operação de comparação ($<$, $>$, $=$, $<=$, $>=$, $\sim=$)

$$R \bowtie_F S = \sigma_F (R \times S)$$

Equi-join(junção de igualdades)

No caso do predicado de F conter apenas igualdades ($=$), teta-join é conhecida como equi-join

Natural Join (junção natural)

A junção natural são uma equi-join entre duas relações R e S sobre todos os atributos comuns x. Uma ocorrência de cada atributo comum é eliminada do resultado. O grau de de junções naturais é a soma dos graus das relações R e S menos o números de atributos em x.

$$R \bowtie S$$

Outer Join (junção exterior)

Algumas vezes quando juntamos duas relações, uma tupla de uma relação pode não conter uma tupla equivalente em outra relação. Uma pessoa pode querer que uma tupla apareça em uma relação mesmo que não exista uma equivalente em outra relação., isso pode ser realizada usando uma operação "outer join".

A junção "outer join" é a junção de tuplas provenientes de R que não possui nenhum valor em comum com as colunas S que estão incluídas na relação resultado. Neste caso os dados perdidos são atribuídos com **null**.

$$R \bowtie S$$

Semi-join

A operação de semi junção "semi-join" define a relação que contem as tuplas de R que participam da junção de R com S.

$$R \bowtie_F S$$

A semi-junção diminui o numero de tuplas que precisam ser apanhadas pela forma de junção com uma peculiaridade particular muito útil para computação de sistemas distribuídos. Ela pode ser re-escrita usando as operações de projeção e de junção.

$$R \bowtie_F S = \Pi_A (R \bowtie_F S)$$

Onde A é um jogo de todos os atributos de R

- **Intersecção.** A operação de interseção consiste em atribuir todas as tuplas que estão em R e S. R e S tem que ser uma união compatível entre si.

$$R \cap S$$

Ela é expressa em termos de atribuição dea operação de diferença.

$$R \cap S = R - (R - S)$$

A operação de divisão consiste no jogo de tuplas de R definidas sobre os atributos C que se igualam a combinação de toda as tuplas em S.

$$R \div S$$

2.2. Calculo Relacional

O calculo relacional não tem nenhuma relação com cálculos diferenciais ou integrais, mas ele adquiriu este nome, de um parente de logica simbólica chamado Calculo Integral. Na álgebra relacional a ordem é sempre especificar explicitamente as expressões e estratégias para avaliação de como o "query" (consulta) é incluída. Não há nenhuma necessidade de descrever como avaliar a questão, apenas especificar que será recuperado ao invés de especificar como será recuperado. Nesta seção falaremos apenas de uma forma superficial dos cálculos relacionais.

O predicado é a avaliação validada de uma função com argumentos, quando a função é provida com vetores de argumentos a função produz uma expressão conhecida como proposição que pode ter como resultado um verdadeiro ou falso.

Por exemplo considere as duas sentenças abaixo:

- Germes de trigo são vendidos no GangStore Alabama.
- Germes de trigo são vendidos mais que Orégano.

As duas sentenças são consideradas proposições porque podem resultar em valores "verdadeiro" ou "falso"; a função na primeira sentença é: "são vendidos no GangStore Alabama", tendo um único argumentos de entrada que é "Germes de trigo". A função da segunda sentença é : "são vendidos mais que" tendo como dois argumentos de entrada "Germes de trigo" e "Orégano".

Se o predicado fosse substituído pela variável X, teríamos x é vendido no GangStore Alabama", deve haver uma gama de valores que se atribuídas a x levarão a sentença a ser um proposição verdadeira e outra gama de valores que levarão a sentença a ser uma proposição falsa; se P fosse o predicado poderíamos escrever a sentença: a Proposição fosse nomeada P. Se P for um predicado, nós podemos escrever: todo x tal que P é verdade para x:

$\{x \mid P(x)\}$

Os predicados podem ser conectados usando conectivos como:

- \wedge (AND)
- \vee (OR)
- \sim (NOT)

Duas formas de calculo relacional são encontrados nas bases de dados relacionais, são chamadas de:

- **Tuplas-orientadas.** Neste tipo de calculo relacional nos estamos interessados em encontrar filas para qual o predicado é verdadeiro, ela usa variáveis de tupla que têm valores que estão dentro do domínio da relação, por exemplo: especificar a variável da tupla E que corresponde a uma relação de empregado.

GAMA DE E QUE É EMPREGADO

Para expressar esta questão "achar todas as tuplas E tal que $P(E)$ é verdadeira", escrevemos assim:

$\{P \mid P(E)\}$

P é chamado de formula. Por exemplo. Por exemplo, Achar o numero, Ultimo nome, primeiro nome e data do contrato de todos os empregados contratados depois de primeiro de Janeiro de 2005, Escrevemos assim:

GAMA DE E É EMPREGADO

$\{E \mid E.dataDeContrato > 'January 1, 2005'\}$

Dois qualificadores são usados com nessa fórmula.

Exemplo: Qualificador existencial (\exists). Este é usado na formula onde deve ser verdade em pelo menos uma instancia.

GAMA DE E É EMPREGADO

$$\exists E \{E.\text{Numero} = 5001\}$$

Isso significa que existe um registro onde o número de empregado é 5001.

- **Qualificador universal (\forall).** Isto é usado em declarações sobre todas instancias existentes na fila.

GAMA DE E É EMPREGADO

$$\forall E \{E.\text{DataContrato} \sim= '15 \text{ Agosto de } 1998'\}$$

Isso significa que para todo registro em que a data de contrato do empregado não é igual a 15 de agosto de 1998.

Uma fórmula bem-formada em cálculo de predicado usa as regras seguintes:

1. Se P é uma formula n -ary(predicado de n argumentos) e t_1, t_2, \dots, t_n são constantes ou variáveis então a formula será $P(t_1, t_2, \dots, t_n)$.
 2. Se t_1 e t_2 são constantes ou qualquer uma variavel do mesmo dominio e Θ um dos operadores ($<, <=, >, >=, =, \sim=$) então a formula será $t_1 \Theta t_2$.
 3. Se F_1 e F_2 são formulas, assim a conjunção delas será $F_1 \wedge F_2$; a disjunção derá $F_1 \vee F_2$, e anegação será $\sim F_1$.
 4. Se F é uma formula com X variáveis livres então $\exists X(F)$ e $\forall X(F)$ também são formulas.
- **Dominio orientado.** Variaveis que são usados para pegar valores de dominio ao inves de relações de tuplas. Se $P(d_1, d_2, \dots, d_n)$ são pos no predicado com variaveis d_1, d_2, \dots, d_n , então

$$\{d_1, d_2, \dots, d_n \mid P(d_1, d_2, \dots, d_n)\}$$

Significa o conjunto de todas as variáveis de domínio para as quais o predicado é verdade, Teste para condição de associação é sempre para um cálculo relacional orientado a dominio, determinando se valores pertencem a uma relação. A expressão $R(x,y)$ avalia se é verdade que se e somente se existem tuplas na relação R com valores x, y para os dois atributos.

$\{\text{UltimoNome}, \text{PrimeiroNome}, \text{DataContrato} \mid \exists \text{DataContrato} (\text{Empregado}(\text{UltimoNome}, \text{PrimeiroNome}, \text{DataContrato}) \wedge \text{DataContrato} > '1 \text{ de Janeiro de } 2005')\}$

A expressão acima é similar a achar todos os empregados que foram contratados depois de primeiro de janeiro de 2005. Para cada atributo é determinado um nome de variável. A condição $(\text{Empregado}(\text{UltimoNome}, \text{PrimeiroNome}, \text{DataContrato}))$, assegura que as colunas estão definidas na relação de empregado.

3. Structured Query Language (SQL)

"Structured Query Language (SQL)" ou seja linguagem estruturada de consultas, é uma linguagem de transformação orientada, isto é, a linguagem é designada para usar nas relações e transformar entradas em saídas. É uma linguagem que permite ao usuário à:

- criar bases de dados e suas estruturas relacionais correspondentes
- Administrar uma base de dados criando, eliminando e alterando os dados.
- Realizar consultas simples e complexas

Também tem dois componentes principais:

1. Linguagem de definição de dados "Data Definition Language" (DDL) para definir estruturas de bases de dados¹
2. Linguagem de Manipulação de dados "Data Manipulation Language" (DML) para recuperar e alterar os dados.

Para o resto deste capítulo nós usaremos JavaDB para aprender SQL. A notação modificada BMF é usada para representar a sintaxe SQL. Os Meta-Símbolos são mostrados aqui.

	Ou "Or". Escolha de um dos termos
[]	Emcapsulamento opcional de itens
*	Curinga de itens que podem repetir uma ou mais vezes. Porem este possui um significado especial para algumas declarações SQL.
{ }	Grupo de itens que podem ser usados da mesma forma como [], ou *
(), ,	Outras pontuações que são parte da sintaxe.

Tabela 1: BMF Meta-símbolos

As palavras chaves são escritas em caixa alta, um exemplo da sintaxe SQL é apresentado:

```
CREATE [ UNIQUE ] INDEX IndexName
ON TableName ( SimpleNomeDEColuna [ , SimpleNomeDEColuna ] * )
```

3.1. Conceito de programação do SQL JavaDB

Java™DB é um sistema de gerenciamento de base dados relacional baseado na linguagem de programação Java e SQL. É uma versão comercial do projeto de base de dados relacional com código aberto da "Apache Software Foundation's" (ASF) chamado de Derby. Essa implementação é um subconjunto do core SQL-92 com algumas características do SQL-99 e do SQL-2003. Esta seção descreve as declarações, as funções embutidas, os tipos de dados, as expressões e as características especiais que JavaDB contém.

3.1.1. Identificadores

Dicionário de objetos são objetos que os desenvolvedores criam como: tabelas, visões, indexes, colunas e constrangimentos que são armazenados na base de dados. JavaDB armazena informações sobre eles na tabela do sistema, mais conhecida como "**as data dictionary**". Os desenvolvedores usam os identificadores para nomear os dicionários de objeto. Um **identifier** (identificador) é a representação dentro da linguagem que possibilita ao usuário desenvolvedor criar em posição as palavras chaves ou comandos.

Existem dois tipos de identificador e cada um deles possui um jogo de regras. Os identificadores

¹

são chamados de identificadores ilimitados e delimitados.

Regras dos identificadores ilimitados

- Eles não estão entre aspas.
- Eles Tem que começar com uma letra e só pode conter letras, traço baixo e dígitos.
- As letras e dígitos permitidos incluem todos os caracteres UNICODE.

Regras de identificadores delimitados

1. Eles só podem conter caracteres entre aspas, onde as aspas não fazem parte do identificador
2. As aspas servem apenas para identificar o começo e o fim do identificador
3. O espaço tomado por um identificador delimitador é insignificante, eles são truncados
4. Derby traduz duas aspa sucessivos dentro de um identificador delimitado como sendo aspas
5. Períodos dentro de um identificador delimitador não são separadores mas sim partes do identificador, Por exemplo "A.B" é o nome do objeto dicionário enquanto "A"."B" é um objeto dicionário qualificado por outro objeto dicionário (como uma coluna chamada "B" da tabela "A")

Identificadores representando objetos dicionário tem que estar em conformidade com as regras dos identificadores SQL-92 e são chamados de SQL92Identifiers. Um SQL92Identifier é um identificador de objeto de dicionário que está em conformidade com as regras do SQL-92.

As declarações do SQL-92 que identifica os objetos de dicionário:

- São limitados em 128 caracteres SQL-92
- São sensíveis a caixa alta e baixa, a mesmos que estejam delimitados por aspas, Neste caso eles serão automaticamente convertidos em caixa Alta pelo sistema
- A menos que estejam entre aspas não são reservados

```
CREATE VIEW SampleView (RECEIVED) AS VALUES 1
-- Nome da view (visão), SampleView, é armazenada no
-- catalogo do sistema
-- as SAMPLEVIEW

CREATE VIEW "AnotherSampleView" (RECEIVED) AS VALUES 1
-- Nome da view, AnotherSampleView, é armazenada no
-- catalogo do sistema, i.e., a forma "caixa baixa" está
--intacta
-- (AnotherSampleView)
```

Alguns objetos de dicionario podem estar contidos dentro de outros objetos, para evitar ambigüidade de objetos de dicionário de mesmo nome mas contem objetos do dicionário separado você precisa qualifica-los. O SQL92Identifier é "dot-separated", i.e. Significa que cada componente é separado do próximo por um ponto. Considere a tabela Employer e Store, ambas possuem uma coluna chamada number, especificar cada coluna coloca-se um período separando a tabela da coluna, assim Employer.number está claramente se referindo a coluna number da tabela Employer

O dicionário de objetos são identificados como se segue:

- Schema
- Table
- Column
- Aliases

- Synonym
- View
- Index
- Constraint
- Cursor
- Trigger
- AuthorizationIdentifier

3.1.2. Palavras Reservadas

Palavras reservadas são construções da linguagem que não podem ser usadas como identificadores pois, possuem um significado especial para JavaDB. Exemplos de palavras reservadas são:

- CREATE
- DROP
- CONNECT
- SELECT
- INSERT

3.1.3. Tipo de dados

Definem quais tipos de valores podem ser armazenados em colunas nas tabelas. As classificações de tipos de dados suportados pelo JavaDB são:

- Tipos de Dados Numéricos
- Tipos de Dados de Caracteres (Char)
- Tipos de dados de Data e Tempo (Date e Time)
- Tipos de dados de Objetos (Large Object)

Detalhes dos tipos de dados do JavaDB são discutidos na lição 4 - Banco de Dados Físico na seção Regras de Negócio e Restrições de Integridade.

4. Criando um Banco de Dados em JavaDB

Nessa seção usaremos a IDE NetBeans para criar um banco de dados para uma Loja Orgânica, chamado organic shop. Assumimos que o JavaDB já está configurado para trabalhar com a IDE NetBeans.

O NetBeans fornece a opção JavaDB Database na opção Tools do menu principal. Nesse item de menu é possível iniciar e parar o servidor, criar uma nova instância de banco de dados, como também registrar servidores de banco de dados na IDE.

4.1. Iniciando o servidor de banco de dados

Escolha Tools > Java DB Database > Start Java DB Server. Observe a seguir a saída na Janela de saída, indicando que o servidor foi iniciado.

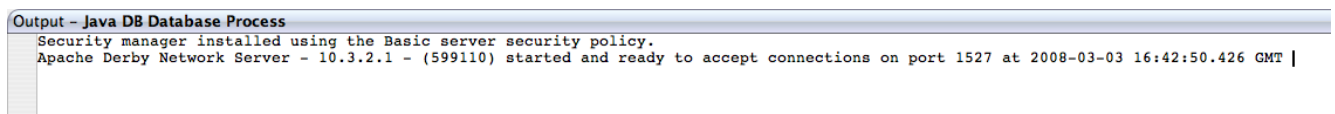


Figura 1: Mensagem inicial do JavaDB

4.2. Criando o banco de dados

Escolha Tools > Java DB Database > Create Database. A janela Create Java DB Database abrirá.

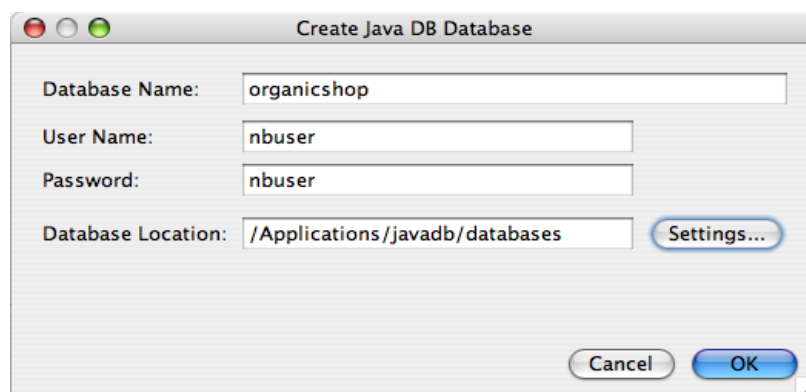


Figure 2: Janela da criação do banco dados Java DB

No campo Database Name, digite **organicshop**. Em User Name e Password digite nbuser para os dois campos. Pressione OK. A tela capturada na Figura 2 mostra o exemplo. A figura 3 mostra o banco de dados organicshop criado na janela Services.

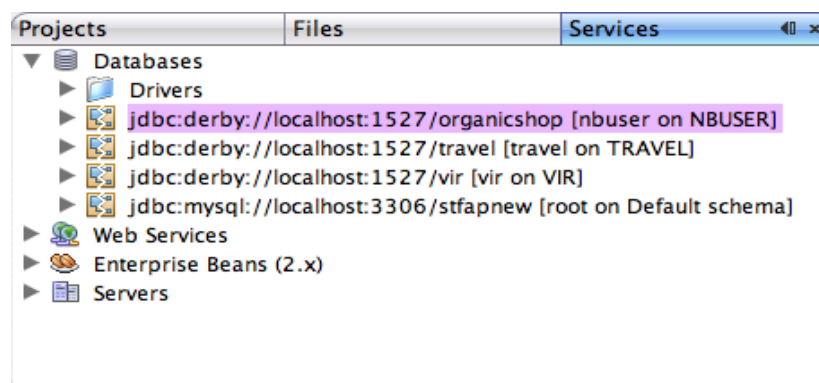


Figure 3: O banco de dados Organic Shop criado

4.3. Conectando o NetBeans ao Banco de Dados

O Database Explorer do NetBeans, disponível na janela Services, fornece funcionalidades para tarefas comuns na manipulação do banco de dados. Isso inclui:

- criação, exclusão e modificação de tabelas
- preenchimento de tabelas com dados
- visualização de dados tabulares
- execução de instruções e consultas SQL

Para começar a trabalhar com o banco de dados, deverá ser estabelecida uma conexão exclusiva. Para conectar com o banco de dados:

1. Abra o Database Explorer na janela Services (Ctrl-5) e localize o novo banco de dados como mostrado na Figura 3.
2. Clique com o botão direito na opção database connection (`jdbc:derby://localhost:1527/organicsshop[nbuser on NBUSER]`) e escolha a opção Connect.
3. Na janela Connect, digite a senha (`nbuser`) e clique em **OK**. Observe que o ícone da conexão agora aparece inteiro, pois a conexão foi estabelecida com sucesso. Quando a conexão não está estabelecida ele aparece quebrado.

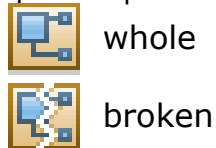


Figura 4: Indicador de Conexão com o banco de dados

4.4. Desconectando o NetBeans do Banco de dados

Clique com o botão direito do mouse na opção:

(`jdbc:derby://localhost:1527/organicsshop[nbuser on NBUSER]`) . Selecione **Disconnect**. Agora o ícone que representa a conexão aparecerá quebrado.

4.5. Excluindo um banco de dados

Com o banco de dados desconectado. Clique com o botão direito do mouse na opção:

(`jdbc:derby://localhost:1527/organicsshop[nbuser on NBUSER]`)

Selecione **Delete**. A caixa de diálogo aparecerá. Clique em **Yes**.

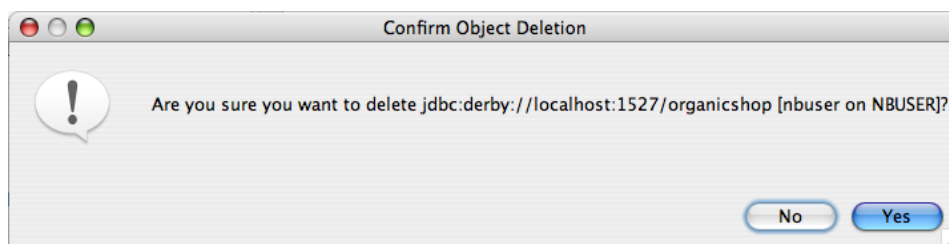


Figura 5: Caixa de Diálogo de Confirmação de Exclusão de Objetos

Na sessão seguinte, usaremos o e NetBeans SQL editor para executar instruções SQL. Para visualizar o SQL Editor, clique com o botão direito do mouse na opção: (`jdbc:derby://localhost:1527/organicsshop[nbuser on NBUSER]`) Selecione a opção Execute. A Figura 6 mostra o SQL Editor.

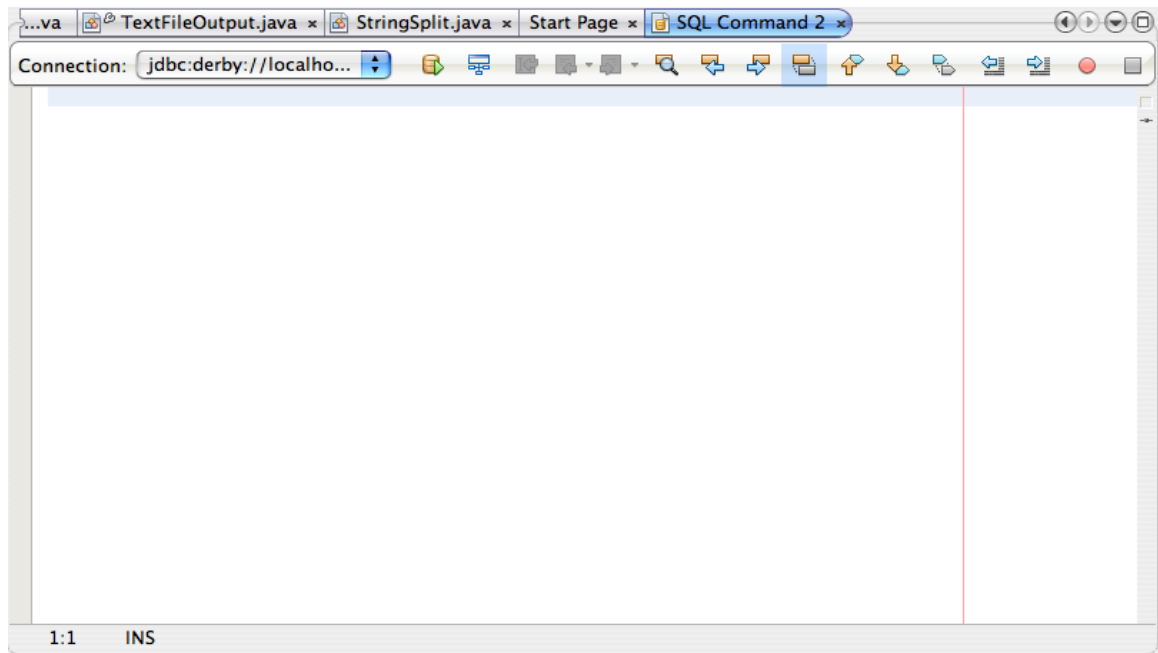


Figura 6: SQL Editor no NetBeans

Para executar o SQL statement simplesmente digite SQL statement no editor. O NetBeans deve estar conectado ao banco de dados. Clique no botão Run SQL. A saída (Output) da instrução é mostrada no Output Pane. A Figura 7 mostra um exemplo.

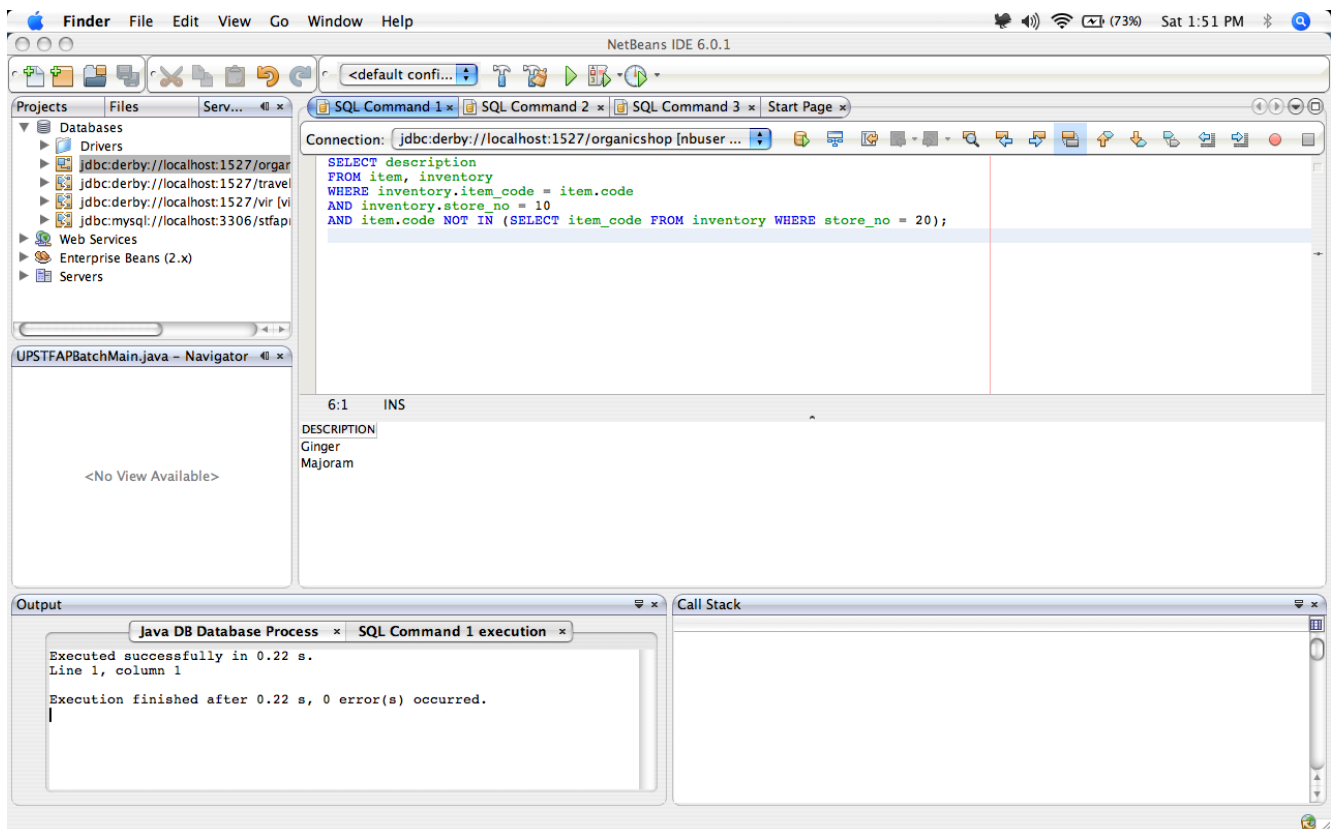


Figura 7: Executando Comandos SQL Statement no NetBeans

5. Linguagem de Definição de Dados

Linguagem de Definição de Dados / Data Definition Language (DDL) são comandos SQL que constroem a estrutura do dicionário de dados (objetos). Eles incluem, criam, alteram e excluem tabelas, visões e índices. Os comandos SQL que são consideradas nessa sessão envolvem o seguintes dicionários de dados:

- Tabelas
- Índices
- Visões

5.1. Comandos de tabela

Os comandos de tabela manipulam a estrutura das tabelas no banco de dados. Nessa sessão analisaremos as seguintes:

- comando CREATE TABLE
- comando ALTER TABLE
- comando RENAME TABLE
- comando DROP TABLE

5.1.1. Comando CREATE TABLE

O comando CREATE TABLE cria uma tabela. Uma tabela contém colunas e restrições. Restrições são regras que os dados devem seguir. Elas podem ser as seguintes:

- Restrição a nível de coluna (column-level), estas referem-se a uma coluna individual em uma tabela, ela não especifica o nome da coluna exceto para verificar as restrições. A Tabela 1 mostra as restrições a nível de coluna e seus significados.

Restrição a nível de coluna	Significado
NOT NULL	Especifica que a coluna não pode receber valores nulos
PRIMARY KEY	Especifica que a coluna identifica unicamente uma linha na tabela
UNIQUE	Especifica que os valores na coluna devem ser únicos. Valores nulos não serão permitidos
FOREIGN KEY	Especifica que os valores de uma coluna devem corresponder aos valores em uma chave primária referenciada ou chave única ou que elas são nulas
CHECK	Especifica regras para os valores na coluna

Tabela 2: Restrição a nível de coluna

- Restrição a nível de tabela refere-se a uma ou mais colunas na tabela. Ela especifica os nomes das colunas a serem aplicadas. A Tabela 2 mostra a lista de comandos a nível de tabela e seus significados.

Restrição	Significado
PRIMARY KEY	Especifica a coluna ou as colunas que identificam como única uma linha na tabela. Valores nulos (NULL) não são permitidos.
UNIQUE	Especifica que os valores nas colunas devem ser únicos. A coluna identificada deve ser definida como não nula (NOT NULL).
FOREIGN KEY	Especifica que os valores nas colunas devem corresponder aos valores na chave primária referenciada ou chave única ou que elas são nulas. Se a chave estrangeira (foreign key) consistir de múltiplas colunas, e alguma coluna for nula, toda a chave será considerada nula. A inserção é permitida desde que não seja em colunas não nulas.

CHECK	Especifica uma ampla gama de regras para os valores em uma tabela.
-------	--

Tabela 3: Restrição a nível de tabela

Os dois tipos de restrição tem a mesma função. A única diferença é onde eles são especificados. Restrições a nível de tabela permitem ao desenvolvedor especificar mais de uma coluna com as restrições PRIMARY KEY, UNIQUE, CHECK ou FOREIGN KEY.

Restrições a nível de coluna referem-se a somente uma coluna, exceto para restrições CHECK. A Sintaxe para o comando CREATE TABLE é mostrada a seguir:

```
CREATE TABLE nome-da-tabela
({{definição da coluna | Restrição a nível de tabela}
[, {definição da coluna | Restrição a nível de tabela} ] * ) |
[(nome-da-coluna [ , nome-da-coluna ] * )]
AS query-expression
WITH NO DATA
}
Definição da coluna:
nome-da-coluna tipo-do-dado
[ Restrição a nível de coluna ]*
[ [ WITH ] DEFAULT { Expressão constante | NULL }
| especificação da coluna gerada ]
[ restrição a nível de coluna ]*
Especificação da coluna gerada:
[ GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ (START WITH IntegerConstant
[, INCREMENT BY IntegerConstant]] ) ] ] ] ]
```

Figura 8 mostra o comando CREATE TABLE para o projeto da tabela HOURLY_EMPLOYEE, no banco de dados The Organic Shop Physical Database.

HOURLY_EMPLOYEE	
Number	Hourly_rate
SMALLINT	DECIMAL
5XXX	999999,99
PK	
5001	250,00

```
CREATE TABLE hourly_employee(
    number SMALLINT PRIMARY KEY,
    hourly_rate DECIMAL(20,2)
);
```

Figura 8: Comandos para criação da tabela HOURLY_EMPLOYEE

Neste exemplo a tabela hourly_employee tem duas colunas; number e hourly_rate. A coluna number tem uma restrição PRIMARY KEY.

Um valor padrão para uma coluna pode ser especificado, ou seja, se um valor não é especificado durante a inserção de uma linha, um valor padrão será assumido para aquela coluna. Como exemplo, considere a criação da tabela salaried_employee da Figura 9.

SALARIED_EMPLOYEE		
Number	Annual_salary	Stock_option
SMALLINT	DECIMAL(30,2)	CHAR(1)
5XXX	999999,99	Y or N
PK		default='N'
5001	546000,00	N

```
CREATE TABLE salaried_employee(
    number SMALLINT PRIMARY KEY,
    annual_salary DECIMAL(30,2),
    stock_option CHAR(1) DEFAULT 'N'
```

);

Figura 9: Coluna com valor padrão e restrições de tabela

Nesse exemplo, para a coluna `stock_option` será colocado o valor padrão (DEFAULT) onde estiver inserido o 'N' quando nenhum valor for especificado para essa coluna.

As colunas podem ter um valor inteiro incrementado quando um registro for inserido na tabela. Tais colunas são conhecidas como **auto-incrementadas ou identity**. Elas comportam-se similarmente aos padrões, mas o valor não é uma constante e sim um valor inserido automaticamente.

Somente as colunas com os seguintes tipos de dados podem ser auto-incrementadas:

- SMALLINT
- INT
- BIGINT

A palavra-chave `IDENTITY` é usada para definir uma coluna como auto-incrementada.

Há dois tipos:

- **GENERATED ALWAYS**

Esse tipo incrementará o valor padrão a cada inserção. Nesse tipo, os valores não podem ser inseridos ou atualizados diretamente. Durante a inserção de registros ou especifica-se a palavra-chave `DEFAULT` ou deixa-se a coluna incrementada completamente fora da inserção na coluna. Como exemplo considere o comando `CREATE TABLE` para a tabela `Store table` na Figura 10.

STORE		
Number	Name	Address
SMALLINT	VARCHAR(250)	VARCHAR(250)
99	X(250)	X(250)
PK,UA,ND1	NN,ND1	
10	GangStore in Alabama	Alabama
20	GangStore in West Virginia	West Virginia
30	GangStore in North Dakota	NorthDakota

```
CREATE TABLE store(
  number SMALLINT GENERATED ALWAYS AS IDENTITY
    (START WITH 10, INCREMENT BY 10),
  name VARCHAR(250) NOT NULL,
  address VARCHAR(250),
  PRIMARY KEY (number)
);
INSERT INTO store VALUES
  (DEFAULT, 'GangStore in Alabama', 'Alabama');
INSERT INTO games (name, address) VALUES
  ('GangStore in West Virginia', 'West Virginia');
```

Figura 10: Auto-incremento `GENERATE ALWAYS`

Nesse exemplo, `number` é definido como um atributo de uma coluna identidade, o qual é um `GENERATED AS`. A primeira linha inserida terá `number` igual a 10 como seu valor inicial. Cada registro inserido, desde então, terá um incremento de 10. O primeiro comando `INSERT` irá atribuir 10 para a coluna `number` da loja do Alabama. O segundo comando `INSERT` irá atribuir o valor 20 para a coluna `number` da loja de West Virginia. O comando `INSERT` será discutido mais adiante. Valores gerados em uma coluna identidade `GENERATE ALWAYS` são únicos. Se os `START WITH` e `INCREMENT BY` não são especificados, o valor inicial será 1, e os incrementos também serão em 1's.

Figura 10 também mostra um exemplo de uma especificação de uma *constraint* de tabela, isto é, a *constraint* de tabela `PRIMARY KEY` na última parte do comando `CREATE TABLE`. Além disso, a *constraint* de coluna `NOT NULL` é especificada para a coluna `name`. Isso significa que quando linhas são inseridas, valores devem ser especificados para essa coluna.

- **GENERATED BY DEFAULT**

Esse é um outro tipo de atributo de coluna identidade. Diferentemente das colunas GENERATED ALWAYS, um valor pode ser especificado na inserção de um comando ao invés do valor gerado por padrão. Para usar o padrão gerado, especifique a palavra-chave DEFAULT quando inserindo na coluna identidade, ou apenas deixe a coluna identidade de fora da lista de inserção em coluna. Para especificar um valor, inclua-o no comando de inserção. Considere o seguinte comando SQL.

```
CREATE TABLE games(
    listNo INTEGER GENERATED BY DEFAULT AS IDENTITY,
    title VARCHAR(150)
);
-- especifica valor "1":
INSERT INTO games VALUES (1, 'Boom or boom');
-- usa padrão gerado
INSERT INTO games VALUES (DEFAULT, 'Capture Ben10');
-- usa padrão gerado
INSERT INTO games(title) VALUES ('Eddy');
```

A coluna GENERATED BY DEFAULT não garante unicidade. Dessa forma, no exemplo acima, as linhas 'Boom or boom' e 'Capture Ben10' irão ambas ter um valor identidade de 1, porque a coluna gerada começa em 1, e o valor especificado pelo usuário também era 1. Para prevenir duplicação, especialmente quando carregando ou importando dados, crie a tabela usando o valor START WITH, o qual corresponde ao primeiro valor identidade que o sistema deve atribuir. Para checar essa condição e desabilitá-la, a *constraint* PRIMARY KEY ou UNIQUE da coluna identidade GENERATED BY DEFAULT deve ser usada.

Atribuir um valor negativo para a cláusula INCREMENT BY irá decrementar o valor em cada inserção. Os valores máximos e mínimos permitidos são dependentes do tipo de dados da coluna. Tentar inserir valores fora do alcance válido do tipo de dados causará uma exceção.

5.1.2. Comando ALTER TABLE

O comando ALTER TABLE modifica a estrutura de uma tabela existente. A sintaxe do comando ALTER TABLE é mostrada a seguir:

```
ALTER TABLE table-Name
{
    ADD COLUMN column-definition |
    ADD CONSTRAINT clause |
    DROP { PRIMARY KEY | FOREIGN KEY constraint-name |
        UNIQUE constraint-name | CHECK constraint-name |
        CONSTRAINT constraint-name }
    ALTER [ COLUMN ] column-alteration |
    LOCKSIZE { ROW | TABLE }
}
```

definição da coluna:

```
Simple-column-NameDataType
[ Column-level-constraint ]*
[ [ WITH ] DEFAULT {ConstantExpression | NULL } ]
```

alteração da coluna:

```
column-Name SET DATA TYPE VARCHAR(integer) |
column-name SET INCREMENT BY integer-constant |
column-name RESTART WITH integer-constant |
column-name [ NOT ] NULL |
column-name [ WITH ] DEFAULT default-value
```

Ele permite ao desenvolvedor:

- **Adicionar uma coluna.** A definição de coluna é similar à definição de coluna do comando CREATE TABLE. *Constraints* de coluna podem ser colocadas em uma nova coluna. Uma coluna com uma *constraint* NOT NULL pode ser adicionada à uma tabela existente se um valor padrão tiver sido fornecido. De outra forma, uma exceção é lançada quando o

comando ALTER TABLE é executado. Se a *constraint* é UNIQUE ou PRIMARY KEY, a coluna não pode conter valores nulos; o atributo NOT NULL deve também ser especificado.

- **Adicionar uma *constraint*.** Uma *constraint* a nível de tabela pode ser adicionada à uma tabela existente, exceto por algumas limitações. Essas limitações são:
 - JavaDB checa a tabela para ter certeza que as linhas satisfazem as *constraints* FOREIGN KEY ou CHECK. Se houver qualquer linha inválida, JavaDB lança uma exceção e a *constraint* não é criada.
 - Todas as colunas incluídas em uma chave primária devem conter dados não nulos e serem únicas.
- **Destruir em uma *constraint* existente.** Deleta uma *constraint* em uma tabela existente. Se a *constraint* não possuir nome, o nome gerado na tabela de sistema SYS.SYSCONSTRAINTS pode ser usado. Quando se deleta uma *constraint* PRIMARY KEY, UNIQUE, ou FOREIGN KEY, se deleta o índice que reforça aquela *constraint*. Isso é conhecido como *backing index*.
- **Modificar uma coluna.** A alteração de coluna permite ao desenvolvedor modificar a definição de coluna da tabela. O desenvolvedor pode:
 - Aumentar o número de caracteres de uma coluna VARCHAR existente; quando a diminuição do tamanho não é permitida ou para modificar o tipo de dados. Além disso, quando modificar a coluna que é parte de uma PRIMARY KEY ou UNIQUE KEY referenciada por uma *constraint* FOREIGN KEY ou parte de uma FOREIGN KEY não é permitido.
 - Especificar o intervalo entre valores consecutivos da coluna identidade.
 - Modificar a *constraint* de nulabilidade. Quando a *constraint* NOT NULL é adicionada à uma coluna existente, valores NULOS não devem existir dentro daquela coluna. Quando a *constraint* NOT NULL é removida, a coluna ou colunas não devem ser usadas em uma *constraint* PRIMARY KEY ou UNIQUE.
- Modificar o valor padrão da coluna.
- Modificar a granularidade da tabela.

A seguir temos alguns exemplos do uso do comando ALTER TABLE.

```
ALTER TABLE games
  ADD COLUMN url VARCHAR(200)
  CONSTRAINT NEW_CONSTRAINT CHECK (url IS NOT NULL);

ALTER TABLE games
  ADD CONSTRAINT title_unique UNIQUE(title);

ALTER TABLE games
  DROP CONSTRAINT title_unique;

ALTER TABLE games
  ALTER COLUMN title SET DATA TYPE VARCHAR(300);
```

O primeiro comando ALTER TABLE adiciona o nome de coluna `url` para a tabela `games` com uma *constraint* de coluna chamada `NEW_CONSTRAINT`, a qual checa se `url` não é nulo. O segundo comando ALTER TABLE adiciona uma *constraint* à tabela `games` chamada `title_unique`, a qual garante que aqueles rótulos serão únicos na tabela. O terceiro comando ALTER TABLE deleta a *constraint* `title_unique`. Por último, o tipo de dados de `title` foi modificado para ser VARCHAR(300) ao invés de VARCHAR(150).

5.1.3. Comando RENAME TABLE

O comando RENAME TABLE modifica o nome da tabela. A sintaxe desse comando SQL é:

```
RENAME TABLE tableName TO newTableName;
```

Se uma *view* ou chave estrangeira referenciar a tabela que está sendo renomeada, um erro é gerado. Se há qualquer *constraint* de checagem ou *triggers* na tabela, um erro é gerado quando tentamos renomear a tabela.

```
RENAME TABLE games TO mygames;
```

A tabela games é modificada para mygames.

5.1.4. Comando DROP TABLE

O comando DROP TABLE elimina a tabela. *Triggers*, *constraints* e índices na tabela serão "destruídos" ou eliminados. A sintaxe é mostrada a seguir:

```
DROP TABLE tableName;
```

O seguinte comando elimina a tabela mygames.

```
DROP TABLE mygames;
```

5.2. Comandos INDEX

Os comandos INDEX permitem criar, modificar e deletar índices de tabelas. JavaDB usa índices para melhorar a performance de comandos de manipulação de dados. Eles incluem:

- Comando CREATE INDEX
- Comando DROP INDEX

5.2.1. Comando CREATE INDEX

O comando CREATE INDEX cria um índice em uma tabela. Índices podem ser sobre uma ou mais colunas na tabela. A sintaxe desse comando é mostrada a seguir:

```
CREATE [UNIQUE] INDEX index-Name  
ON table-Name ( Simple-column-Name [ ASC | DESC ]  
[ , Simple-column-Name [ ASC | DESC ] ] * )
```

O número máximo de colunas para uma chave de índice é 16. O nome do índice é limitado em 128 caracteres. Um nome de coluna apenas deve ser especificado uma vez em um único comando CREATE INDEX. Entretanto, diferentes índices podem usar a mesma coluna. A palavra-chave DESC ordenada a coluna em ordem decrescente. Por padrão, índices são ordenados em ordem crescente. As *constraints* UNIQUE, PRIMARY e FOREIGN KEY geram índices para frente ou para trás na *constraint* (também conhecidos como *backing indexes*). Se uma coluna ou um conjunto de colunas possuem uma *constraint* UNIQUE ou PRIMARY KEY, o sistema irá automaticamente criar o índice usando um nome gerado por ele. Adicionando uma *constraint* PRIMARY KEY ou UNIQUE quando um índice UNIQUE existente aponta para o mesmo conjunto de colunas irá resultar em dois índices físicos na tabela para o mesmo conjunto de colunas. Um dos índices é o índice original UNIQUE e o outro é o *backing index* para a nova *constraint*. Vejamos alguns exemplos a seguir:

```
CREATE INDEX description_idx ON item (description);  
CREATE INDEX storeAddr_idx ON STORE (address DESC);
```

O primeiro comando CREATE INDEX cria um índice na tabela PRODUCT na descrição de coluna chamada description_idx. A descrição nesse índice é ordenada na ordem crescente. O segundo comando CREATE INDEX cria um índice na tabela STORE na coluna address chamado storeAddr_idx. Endereços são ordenados em ordem decrescente porque a palavra-chave DESC é usada.

5.2.2. Comando DROP INDEX

O comando DROP INDEX deleta ou "dropa" um índice. A sintaxe é mostrada a seguir:

```
DROP INDEX indexName;
```

O exemplo a seguir elimina description_idx da tabela PRODUCT e storeAddr_idx da tabela STORE.

```
DROP INDEX description_idx;  
DROP INDEX storeAddr_idx;
```

5.3. Comandos de Views

São tabelas virtuais formados por uma *query*. É um objeto de dicionário que pode ser usado antes de ser "destruído". Não são atualizáveis. Os comandos para gerenciar Views discutidos nessa seção são:

- Comando CREATE VIEW
- Comando DROP VIEW

5.3.1. Comando CREATE VIEW

O comando CREATE VIEW cria uma *view* baseada no comando SELECT. O comando SELECT será discutido na seção seguinte deste capítulo. A sintaxe é mostrada a seguir:

```
CREATE VIEW view-Name  
    [ ( Simple-column-Name [, Simple-column-Name] * ) ]  
AS Query
```

Um exemplo de uma definição simples de uma *view* é mostrado a seguir. ListOfItemView mostrará o nome da loja (store), o nome do item e a quantidade restante.

```
CREATE VIEW ListOfItemView AS  
    SELECT store.name, item.description, quantity  
    FROM store, item, inventory  
    WHERE store.number = inventory.store_no  
    AND item.code = inventory.item_code;
```

Uma definição de *view* pode conter uma *view* opcional chamada *column list* para explicitamente nomear as colunas na *view*. Se não houver nenhuma *column list*, a *view* herda os nomes de coluna da *query* base. Todas as colunas em uma *view* devem ter nomes únicos. Como exemplo, considere o comando CREATE VIEW a seguir. SampleBonusView possui duas colunas COL_SUM, as quais são computadas como a soma de *commission* e *bonus*, e COL_DIFF, o qual é computado como a diferença entre *commission* e *bonus*.

```
CREATE VIEW SampleBonusView (COL_SUM, COL_DIFF) AS  
    SELECT COMM + BONUS, COMM - BONUS  
    FROM employee;
```

5.3.2. Comando DROP VIEW

O comando DROP VIEW elimina ou "destrói" uma *view*. A sintaxe é mostrada a seguir:

```
DROP VIEW view-Name;
```

No exemplo mostrado a seguir. A SampleBonusView é eliminada da base de dados. Qualquer comando que referencie a *view* será invalidado. DROP VIEW não é permitido se há outras *views* que o referenciam.

```
DROP VIEW SampleBonusView;
```

5.4. Triggers

São mecanismos que residem na base de dados. Eles implementam as operações de gatilho que forem identificadas durante o projeto físico da base de dados. Como revisão, operações de gatilho são uma declaração ou regra que governa a validade das operações de manipulação de dados, como INSERT, UPDATE e DELETE.

Há dois componentes de *triggers*. São eles:

- **Evento de gatilho**, o qual especifica qual evento ocorreu em uma tabela, caracterizado pelos comandos de manipulação de dados INSERT, UPDATE ou DELETE linhas; e

- **Ação de Gatilho**, o qual especifica o que precisa ser executado.

A sintaxe para a criação de *triggers* é mostrado na figura a seguir.

```
CREATE TRIGGER TriggerName
{ AFTER | NO CASCADE BEFORE }
{ INSERT | DELETE | UPDATE [ OF column-Name [, column-Name]* ] }
ON table-Name
[ ReferencingClause ]
[ FOR EACH { ROW | STATEMENT } ] [ MODE DB2SQL ]
Triggered-SQL-statement

ReferencingClause:
{
{ OLD | NEW } [ AS ] correlation-Name [ { OLD | NEW } [ AS ]
correlation-Name ] |
{ OLD_TABLE | NEW_TABLE } [ AS ] Identifier [ { OLD_TABLE | NEW_TABLE }
[AS] Identifier ]
}
```

O evento de gatilho é um tipo de comando que ativa um *trigger*, que pode ser:

- Comando INSERT
- Comando DELETE
- Comando UPDATE

As ações *trigger*, por outro lado, são comando SQL que são executados quando o evento de gatilho ocorre, o qual pode ser também:

- Comando INSERT
- Comando DELETE
- Comando UPDATE

A ação de gatilho deve também especificar quando o *trigger* deve ser executado. Pode ser:

- *BEFORE Triggers* executam antes das modificações de comando serem aplicadas e antes de quaisquer *constraints* serem aplicadas. É executado somente uma vez.
- *AFTER Triggers* executam depois que todas as *constraints* são satisfeitas e as modificações são aplicadas à tabela alvo. É executado somente uma vez.
- *FOR EACH Triggers*, também conhecido como *triggers* de linha, executam para cada linha afetada pelo evento de gatilho. Se nenhuma linha é afetada, não é executado. É definido pelo uso das palavras-chave FOR EACH.

A cláusula REFERENCING permite a criação de dois prefixos que são combinados com o nome da coluna, um para referenciar o valor antigo, e o outro para referenciar o novo valor. Muitos *triggers* necessitam referenciar o dado que está sendo atualmente modificado pelo evento da base de dados que ocasionou suas execuções. Eles podem precisar referenciar os novos valores.

A palavra-chave OLD significa que o prefixo irá referenciar valores antigos. A palavra-chave NEW significa que o prefixo irá referenciar novos valores.

A ação de gatilho definida pelo *trigger* é chamada de comando *trigger* SQL. Para JavaDB, há algumas limitações:

- Não deve conter qualquer parâmetro dinâmico.
- Não deve criar, alterar ou "destruir" a tabela sobre a qual o *trigger* é definido.
- Não deve adicionar ou remover um índice da tabela na qual o *trigger* é definido.
- Não deve adicionar ou "destruir" uma *trigger* da tabela sobre a qual o *trigger* é definido.
- Não deve ocorrer *commit* ou *rollback* sobre a transação corrente ou modificação do nível de isolamento. Mais detalhes no Capítulo Gerenciamento de Transações.
- *Triggers Before* não podem ter comandos INSERT, UPDATE ou DELETE como suas ações.
- *Triggers Before* não podem chamar procedimentos que modifiquem dados SQL como suas

ações.

Um exemplo de criação de um *trigger* e sua execução implícita é mostrado a seguir.

```
CREATE TABLE salary_update_log(  
    change_date DATE,  
    emp_number SMALLINT,  
    old_salary DECIMAL(20,2),  
    new_salary DECIMAL(20,2)  
);  
  
CREATE TRIGGER audit_hourly_rate  
    AFTER UPDATE OF hourly_rate ON hourly_employee  
    REFERENCING OLD AS OLD_VALUE NEW AS NEW_VALUE  
    FOR EACH ROW  
    INSERT INTO salary_update_log VALUES  
        (CURRENT_DATE, OLD_VALUE.number,  
         OLD_VALUE.hourly_rate, NEW_VALUE.hourly_rate);
```

A *trigger* `audit_hourly_rate` é um exemplo de *trigger* de auditoria que mantém registro das alterações feitas para o valor de `hourly_rate` para qualquer registro da tabela `hourly_employee`. Para esta *trigger*, a tabela `salary_update_log` precisa ser criada; é onde as alterações são registradas em log. A ação da *trigger* é uma instrução `INSERT` na tabela `salary_update_log` onde os valores são as datas onde as datas foram realizadas (`change_date`), o registro do empregado que foi afetado (`emp_number`), o tarifa antiga de hora (`OLD_VALUE.hourly_rate`) e a nova tarifa (`NEW_VALUE.hourly_rate`).

```
UPDATE hourly_employee SET  
    hourly_rate = hourly_rate + 5;  
  
SELECT * FROM salary_update_log;
```

Então a *trigger* `audit_hourly_rate` é definida de acordo com as alterações feitas na coluna `hourly_rate` da tabela `hourly_employee`, a instrução `UPDATE` irá automaticamente executar a instrução `INSERT` da *trigger*. A ação disparada é executada depois das alterações terem sido realizadas em cada linha afetada. As alterações são registradas em log em `salary_update_log`. Para remover uma *trigger* do banco de dados, utiliza-se a instrução `DROP TRIGGER`. A sintaxe é mostrada a seguir.

```
DROP TRIGGER triggerName;
```

Quando uma tabela é eliminada, todas as *triggers* associadas são também automaticamente eliminadas. Não é necessário eliminar antes as *triggers* para depois eliminar a tabela. Um exemplo de *trigger* de remoção é mostrada a seguir.

```
DROP TRIGGER audit_hourly_rate;
```

6. Linguagem de Manipulação de Dados (DML)

Linguagem de Manipulação de Dados (DML) são instruções que manipulam dados dentro de tabelas. São frequentemente utilizados em instruções SQL. Consistem em:

- instrução SELECT
- instrução INSERT
- instrução DELETE
- instrução UPDATE

6.1. Instrução *SELECT*

A instrução *SELECT* recupera um ou mais registros de uma ou mais tabelas. Ele cria uma tabela virtual baseada na existência de outras tabelas e restrições incorporadas à estas tabelas. São as instruções mais frequentemente utilizadas na DML. Nesta seção, a sintaxe do *SELECT* é descrita primeiro. Em seguida veremos as operações relacionais que podem ser feitas com a instrução *SELECT*.

A sintaxe é exibida a seguir:

```
SELECT select-clause FROM from-clause
    [WHERE where-clause]
    [GROUP BY group-by-clause]
    [HAVING having-clause]
    [ORDER BY order-by-clause]
```

Para esta seção, a sintaxe da instrução *SELECT* será discutida primeiro. Ela é seguida pelas operações relacionais que podem ser executadas.

A instrução *SELECT* é dividida em (6) cláusulas. São as que seguem abaixo:

1. A cláusula *select* é utilizada para especificar os dados que se quer recuperar de uma ou mais tabelas no banco de dados. Esta é uma cláusula obrigatória e sua sintaxe é exibida a seguir.

```
SELECT [ALL|DISTINCT] select-list

select-list:

{ * | {tablename | alias}.column-name |
  Expression
}
```

Contém uma lista de expressões e um quantificador opcional que é aplicado aos resultados da cláusula de *FROM* e cláusula de *WHERE*. Se a palavra-chave *DISTINCT* for usada, só uma cópia de qualquer valor de linha está incluída no resultado. Se palavra-chave *ALL* for usada, todos os registros são devolvidos como a parte do resultado.

2. A cláusula *from* é utilizada para especificar a(s) tabela(s) ou view(s) da qual os dados virão. Também é uma cláusula obrigatória e sua sintaxe é mostrada a seguir:

```
FROM {tablename | viewname | JOIN Operation}
```

A operação *JOIN* será discutida posteriormente à parte de operações relacionais da instrução *SELECT*.

3. A cláusula *where* é utilizada para especificar uma coleção de uma ou mais condições que farão o uso de operadores lógico e relacionais. Somente linhas para as quais o resultado da avaliação é *TRUE* são retornadas ao resultado. A sintaxe é exibida a seguir:

```
WHERE BooleanExpression
```

Uma expressão booleana (*BooleanExpression*) pode incluir operadores relacionais e lógicos. A

Tabela 4 mostra uma lista de Operadores SQL Booleanos suportados pelo JavaDB.

Operador	Explicação e Exemplo	Sintaxe
AND OR NOT	São operadores lógicos que combina expressões lógicas. Exemplo: (airportCode = 'BRU') OR (airportCode = 'PHP')	{ expressão AND expressão expressão OR expressão NOT expressão }
< = > <= >= <>	São operadores relacionais que são aplicáveis a todos os tipos de dados nativos. Exemplo: DATE('2006-02-26') < DATE('2006-03-01') -- retorna verdadeiro	{ < = > <= >= <> }
IS NULL IS NOT NULL	Utilizados para testar quando o resultado de uma expressão é NULL ou não. Exemplo: WHERE MiddleName IS NULL	{ expressão IS [NOT] NULL }
LIKE	Este operador tenta combinar uma expressão de caracteres a um padrão de caracteres. Padrão de caracteres são strings que incluem caracteres coringas. Caracteres coringas são: <ul style="list-style-type: none"> • % (percentual) que combina qualquer (zero ou mais) número de caracteres na posição correspondente • _ (underscore) que combina um caracter à posição correspondente. Qualquer outro caracter combina somente com o caracter da posição correspondente. Exemplo: ultimonome LIKE 'R%' primeironome LIKE 'Sant_' endereco LIKE '%=_ ' ESCAPE '='	expressãoCaractere s [NOT] LIKE expressãoCaractere sComCoringa [ESCAPE 'escapeChar']
BETWEEN	Este operador é utilizado para testar se o primeiro operador está entre o segundo e o terceiro operandos. O segundo operando deve ser menor que o terceiro operando. Este operador é aplicável somente em tipos de dados que pode-se aplicar <= e >=. Exemplo: WHERE date_hired BETWEEN DATE('2006-02-26') AND DATE('2006-03-01')	expressão [NOT] BETWEEN expressão AND expressão
IN	Este operador funciona em sub-consultas de tabelas ou	{

Operador	Explicação e Exemplo	Sintaxe
	<p>lista de valores. Retorna TRUE se o valor da expressão esquerda está no resultado da sub-consulta da tabela ou na lista de valores. A sub-consulta da tabela pode devolver múltiplas linhas mas deve devolver uma coluna única. Exemplo:</p> <pre>WHERE date_hired NOT IN (SELECT date_hired FROM employee WHERE Store = 10)</pre>	<pre>expressão [NOT] IN subConsultaDa Tabela expressão [NOT] IN (expressão [, expressão]*) }</pre>
EXISTS	<p>Este operador age sobre uma sub-consulta de tabela. Retorna TRUE se a sub-consulta da tabela devolve alguma linha. Caso contrário, retorna FALSE. Exemplo:</p> <pre>WHERE EXISTS (SELECT * FROM employees WHERE date_hired > DATE('2001-01-01'))</pre>	<pre>[NOT] EXISTS SubConsultaDaTab ela</pre>
ALL ANY SOME	<p>Esses são conhecidos como comparação quantificada. Uma comparação quantificada é umas operações lógicas (<, =, >, <=, >=, <>) com ALL ou ANY ou SOME aplicados.</p> <p>Se ALL for usado, a comparação deve ser verdadeira para todos os valores devolvidos pela sub-consulta da tabela.</p> <p>Se ANY ou SOME forem usados, a comparação deve ser verdadeira para pelo menos um valor da sub-consulta da tabela. Example:</p> <pre>WHERE ave_qty < ALL (SELECT amount/500 FROM Inventory)</pre>	<pre>expressão operadorDeComparaç ão { ALL ANY SOME } subConsultaDaTabel a</pre>

Tabela 4: Operadores SQL Booleanos

4. A cláusula **group-by** é utilizado para produzir uma única linha ou resultados para cada grupo. Um **group** é um conjunto de linhas que têm um mesmo valor para cada coluna na cláusula SELECT. É tipicamente utilizada com funções de agregação ou conjunção. Fornece um meio de avaliar expressões sobre um conjunto de linhas. Opera sobre um conjunto de valores e os reduz a uma simples escala de valores. As funções de agregação mais comumente utilizadas são mostradas na Tabela 5.

Função de Agregação	Descrição
COUNT	<p>É uma função de agregação que conta o número de linhas acessadas na expressão. É permitida em todos os tipos de expressões. Sintaxe:</p> <pre>COUNT([DISTINCT ALL] expression)</pre>
MIN	<p>É uma função de agregação que retorna o valor mínimo de uma expressão sobre um conjunto de linhas. É permitida nas expressões que avaliam tipos de dados nativos como CHAR, VARCHAR, DATE, TIME, etc. Sintaxe:</p> <pre>MIN([DISTINCT ALL] expression)</pre>
MAX	<p>É uma função de agregação que retorna o valor máximo de uma expressão sobre um conjunto de linhas. É permitida nas expressões que avaliam tipos de dados nativos como CHAR, VARCHAR, DATE, TIME, etc. Syntax:</p> <pre>MIN([DISTINCT ALL] expression)</pre>

Função de Agregação	Descrição
AVG	É uma função de agregação que avalia a média de uma expressão sobre um conjunto de linhas. É somente permitida sobre expressões de tipos de dados numéricos. Sintaxe: AVG([DISTINCT ALL] expression)
SUM	É uma função de agregação que avalia a soma de uma expressão sobre um conjunto de linhas. É permitida somente sobre expressões que avaliam tipos de dados numéricos. Sintaxe: SUM ([DISTINCT ALL] Expression)

Tabela 5: Funções de Agregação

A sintaxe da cláusula GROUP BY é mostrada a seguir:

```
GROUP BY column-Name [, column-name]*
```

A coluna ou colunas devem estar dentro do escopo da consulta.

5. A cláusula having é utilizada para aplicar um ou mais condições de qualificadores a um grupo. É normalmente associado a uma cláusula GROUP BY. Restringe o resultado de uma especificação GROUP BY. É aplicada a cada grupo de tabelas agrupadas. A sintaxe é demonstrada a seguir:

```
HAVING searchCondition
```

6. A cláusula order-by é utilizada para ordenar o resultado de uma consulta pelos valores contidos em uma ou mais colunas. Especifica uma uma ordem na qual as linhas irão aparecer no resultado. A sintaxe é exibida a seguir:

```
ORDER BY {column-Name | columnPosition | expression} [ASC | DESC]
[, column-Name | columnPosition | expression] [ASC | DESC]]*
```

6.1.1. Operação Relacional da instrução SELECT

Uma relação operacional envolve manipulação de uma ou mais tabelas para produzir um conjunto de registros. É utilizada para aplicar a linguagem relacional que permite manipular registros em uma relação. Existem diferentes tipos de operações relacionais que uma instrução SELECT pode executar.

- Seleção
- Projeção
- Produto Cartesiano
- Junções
- Operações de Conjunções
- Sub-consultas

1. **Seleção.** Uma seleção é feita tomando o subconjunto horizontal de linhas de uma única tabela que satisfazem uma determinada condição. Ele devolve algumas linhas e todas as colunas de uma tabela. Exemplos são exibidos a seguir:

```
SELECT * FROM employee WHERE date_hire < DATE('1998-01-01');
SELECT * FROM employee WHERE (lastname like 'C%') OR (lastname like 'B%');
SELECT * FROM employee WHERE manager IS NULL;
```

O asterisco (*) na cláusula SELECT significa todas as colunas. A primeira instrução SELECT retorna todos os empregados que foram demitidos antes de 1o. de Janeiro de 2008. The first SELECT-statement returns all employees who were hired earlier than January 01, 1998. Isto irá

retornar o registro de Stone.

A segunda instrução SELECT retornará todos os empregados que tem o último nome iniciado com a letra C ou B. Isto irá retornar os registros de Blake, Choo e Cruz.

A última instrução SELECT retornará todos os empregados que não tem nenhum gerente. Isto retornará o registro de Stone.

2. **Projeção.** Uma projeção é realizada tomando o subconjunto vertical das colunas de uma única tabela retendo as linhas únicas. Retorna algumas colunas e todas as linhas de uma tabela. Exemplos são exibidos a seguir:

```
SELECT number, lastname, firstname FROM employees;  
SELECT name FROM store;
```

A primeira instrução SELECT mostra o number, lastname e firstname de todos os empregados da tabela employee.

A segunda instrução SELECT mostra o name de todas as lojas da tabela store.

3. **Produto Cartesiano.** É uma consulta que não demonstra explicitamente uma condição de junção entre as tabelas onde o resultado se compõe de cada combinação possível de linhas das tabelas. O resultado é muito grande, improdutivo e de dados inexatos. Um exemplo é demonstrado em Error: Reference source not found onde todas as linhas da tabela employee são combinadas com todas as linhas da tabela store, o que não faz sentido. Tente evitar os produtos cartesianos.

```
SELECT * FROM employee, store;
```

4. **Junção.** Esta é uma operação pronta. Ocorre quando duas ou mais tabelas são conectadas baseadas na relação entre uma ou mais colunas de cada tabela. Há vários tipos de junções que podem ser criadas:

- **Equi-junção e Junção Natural.** É uma estrutura de consulta na direção que os registros da tabela unida devem ter os mesmos valores das colunas às quais eles são juntados. Podem ser exibidos dados redundantes. Junção Natural, por outro lado, é uma consulta estruturada que não expõe dados redundantemente. Outro termo utilizado para se referir à equi-junção ou junção natural é **inner join**. A figura 43 mostra exemplos de operações de inner join.

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.

Banco do Brasil

Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec

Suporte e apoio financeiro e logístico a todo o processo.

Borland

Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB

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

Módulo 9

Banco de Dados



Lição 6

Java Database Connectivity (JDBC)

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

1. Objetivos

JDBC é o acrônimo para Java Database Connectivity. Trata-se de uma API Java que permite a programas escritos em Java executarem instruções SQL. Esta API permite que programas Java possam interagir com qualquer bando de dados compatível com SQL. Uma vez que todos os sistemas gerenciadores de bancos de dados relacionais (DBMSs) suportam SQL e pelo fato de Java rodar na maioria das plataformas, com JDBC é possível escrever uma simples aplicação que rode em diferentes plataformas e interaja com diferentes DBMSs. Esta lição compreende os seguintes tópicos:

- JDBC Design Pattern
- Implementando JDBC Design Patterns
- Programação de Bancos de Dados server-side
 - Stored Procedures com JDBC
 - Stored Functions com JDBC

Ao final desta lição, o estudante será capaz de:

- Utilizar a Java Database Connectivity (JDBC)
- Entender como utilizar a JDBC Design Pattern
- Utilizado a API Java que permite programas Java a execução de comandos SQL
- Implementar uma aplicação de inventário que manipula registros da tabela de inventário
- Utilizar a JavaDB como banco de dados
- Como criar procedimentos armazenados e funções como programações de servidor de banco de dados

2. JDBC Design Pattern¹

Esta sessão discute o padrão de uso do mecanismo de persistência escolhido para o Sistema de Gerenciamento de Bancos de Dados Relacionais (RDBMS) que é o Java Database Connectivity (JDBC) Design Pattern. Existem duas visões para este pattern, chamadas: **static view** e **dynamic view**.

2.1. Visão Estática do Design Pattern de Persistência

A visão estática é um modelo de objeto do pattern. Isto é ilustrado usando um diagrama de classes. A Figura 1 mostra o design pattern para as classes persistentes.

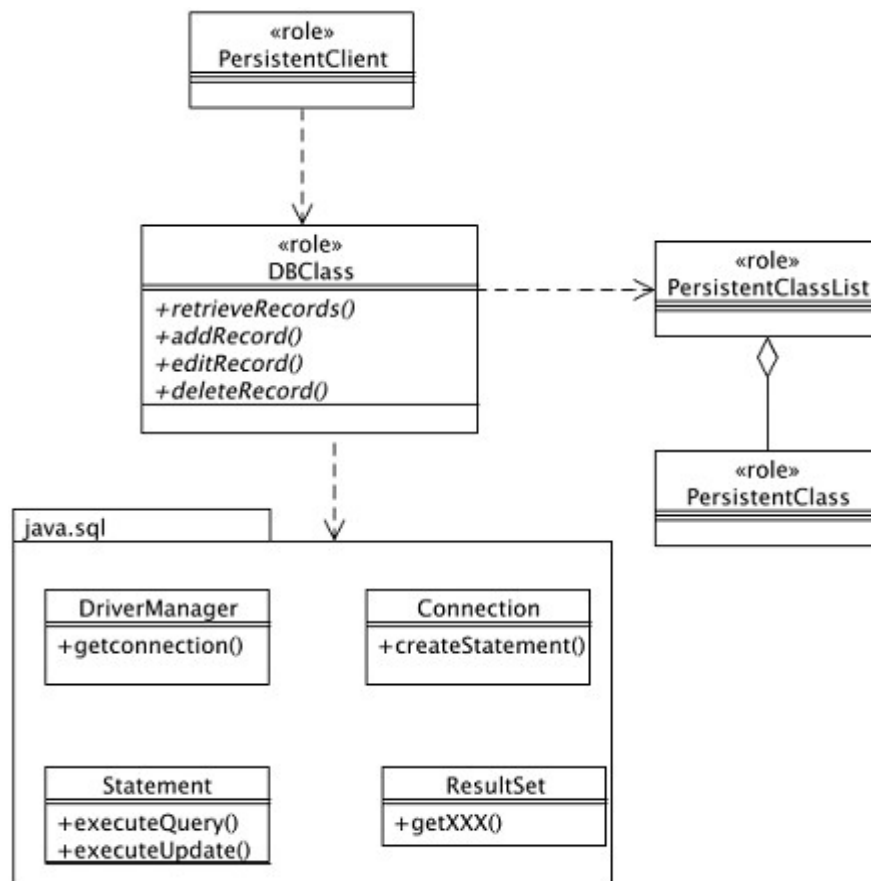


Figura 1: JDBC Design Pattern Visão Estática

Tabela 1 mostra as classes que são definidas pelo programador.

Classe	Descrição
PersistentClient	É uma classe de requisição de dados do banco de dados. Normalmente, são classes de controle perguntando algo a partir de uma entity class. Ela trabalha como uma DBClass .
DBClass	É uma classe responsável pela comunicação com o banco de dados. Trabalha em conjunto com as classes do pacote <code>java.sql</code> .
PersistentClassList	É uma classe usada para retornar um conjunto de objetos persistentes com resultado de uma consulta ao banco de dados. Um registro é equivalente a uma PersistentClass nesta lista.

¹ A sessão do JDBC Design Pattern foi copiada do JEDI-Software Engineering Courseware. Seu uso é de conhecimento do autor.

Classe	Descrição
PersistentClass	É uma classe que mapeia um registro no banco de dados.

Tabela 1: JDBC Design Pattern Classes

A **DBClass** é responsável por tornar outras instâncias de classes persistentes. Entendemos por isso como o mapeamento OO-para-RDBMS. Esta classe estabelece uma interface com o RDBMS. **Toda classe que precisar ser persistida deverá ter uma DBClass correspondente!**

Tabela 2 mostra as classes que estão definidas no pacote java.sql.

Classes	Método	Descrição do Método
Class	forName()	Carrega o driver apropriado para o banco de dados.
DriverManager - É uma classe que permite a manipulação do driver JDBC que foi carregado.	getConnection()	Estabelece uma conexão com o banco de dados alvo. Quando conectado com sucesso, este método retorna um objeto Connection. Caso contrário, uma SQLException é levantada.
Connection É uma classe que representa uma conexão para o banco de dados.	createStatement()	Retorna um objeto Statement.
	close()	Fecha uma conexão.
Statement É uma classe que representa uma instrução SQL. PreparedStatement e CallableStatement são usadas para executar stored procedures e functions.	executeQuery()	Requisita o banco para executar uma instrução SELECT-statement (query). Retorna um objeto ResultSet contendo todas as linhas que satisfazem a SELECT-statement. Caso contrário, retorna null.
	executeUpdate()	Requisita o banco de dados para executar tanto uma instrução INSERT, UPDATE ou DELETE. Pode também executar Data Definition Language como instruções ALTER-statements.
	close()	Fecha uma statement.
ResultSet É uma classe que representa o conjunto de resultados, i.e., o conjunto de registros retornados pela consulta. Existem outros métodos getXXX() para diferentes tipos de dados primitivos usados em Java.	hasNext()	Retorna um valor booleano. Se ainda existirem registros no ResultSet, retorna true. Caso contrário retorna false.
	next()	Retorna o próximo registro do ResultSet.
	getString()	Recupera o valor String da coluna especificada assim como referenciado pelo número que representa a posição do valor dentro do registro.
	getInt()	Recupera o valor inteiro da coluna especificada assim como referenciado pelo número que representa a posição do valor dentro do registro.
	getLong()	Recupera o valor inteiro longo da coluna especificada assim como referenciado pelo número que representa a posição do valor dentro do registro.
	close()	Fecha o resultset.

Tabela 2: Classes comuns do pacote java.sql

Note que as classes Connection, Statement, e ResultSet possuem um método close(). Isto é bom

para deixar claro quem está sendo fechado. Estas classes são alocadas nos recursos do sistema quando são criadas e usadas. Para liberar recursos do sistema, precisamos explicitamente fechá-las, invocando o método `close()`.

O pacote `java.sql` é usado porque contém classes e interfaces para manipulação de bancos de dados relacionais em Java.

2.2. Visão Dinâmica do Padrão de Projeto de Persistência

A visão dinâmica do Padrão de Projeto de Persistência mostra como as classes a partir da visão estática interagem umas com as outras. O diagrama de sequência é usado para ilustrar este comportamento dinâmico. Existem alguns comportamentos dinâmicos que podem ser vistos neste padrão, especificamente, são eles: **JDBC Initialization**, **JDBC Create**, **JDBC Read**, **JDBC Update** e **JDBC Delete**.

1. **JDBC Initialization**. A inicialização deve ocorrer antes que qualquer classe persistente possa ser acessada. A principal necessidade é estabelecer conexão com o RDBMS. Isto envolve o seguinte:

- Carregar apropriadamente o *driver*
- Conexão com o banco de dados

O diagrama de sequência da inicialização do JDBC é exibido na Figura 2.

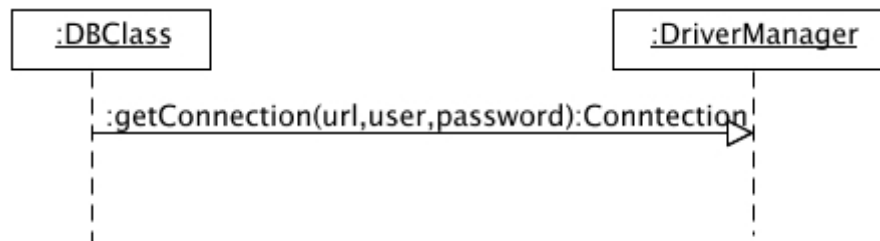


Figura 2: Inicialização do JDBC

A `DBClass` carrega o *driver* apropriado chamando `getConnection(url, user, password)`. Este método estabelece uma conexão com o banco de dados informado na URL de conexão. O `DriverManager` seleciona o *driver* apropriado a partir de um conjunto de drivers JDBC registrados.

2. **JDBC Create**. Este comportamento cria um registro. Executa a instrução SQL `INSERT`. Ele considera que a conexão já esteja estabelecida. O diagrama de sequência é mostrado na Figure 3.

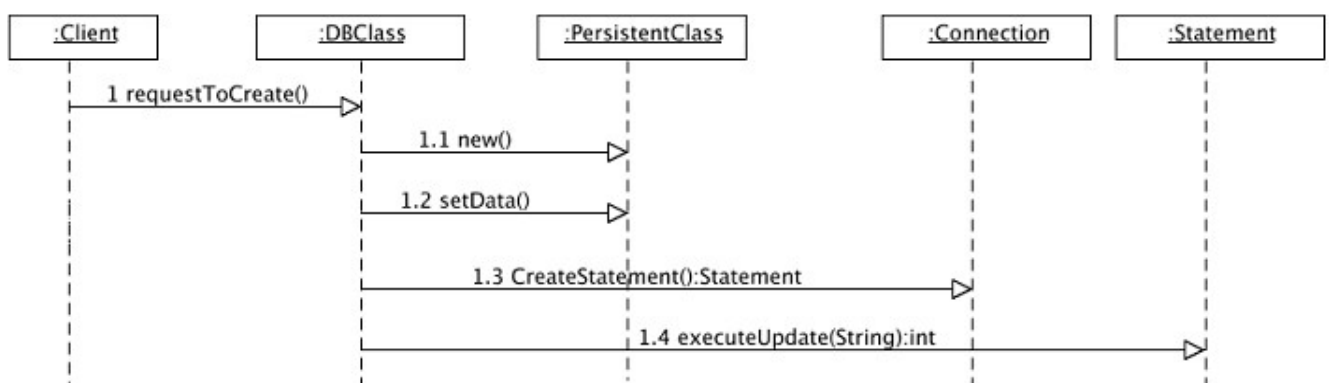


Figure 3: JDBC Create

Descrição do Fluxo do Diagrama de Sequência:

- A **PersistencyClient** pergunta solicita à **DBClass** que crie uma nova classe.
- A **DBClass** cria uma nova instância de **PersistentClass** (`new()`) e atribui valores para a **PersistentClass**.

- A **DBClass**, então, cria uma nova declaração usando **createStatement()** de **connection**. Normalmente, trata-se de uma instrução INSERT em SQL.
- A **Statement** é executada por meio do método **executeUpdate(String):int**. Um registro é inserido no banco de dados.

3. **JDBC Read**. Este comportamento recupera registros do banco de dados. Ele executa a instrução SELECT. Ele também presume que uma conexão com o banco já esteja estabelecida. O diagrama de sequência é mostrado na Figura 4.

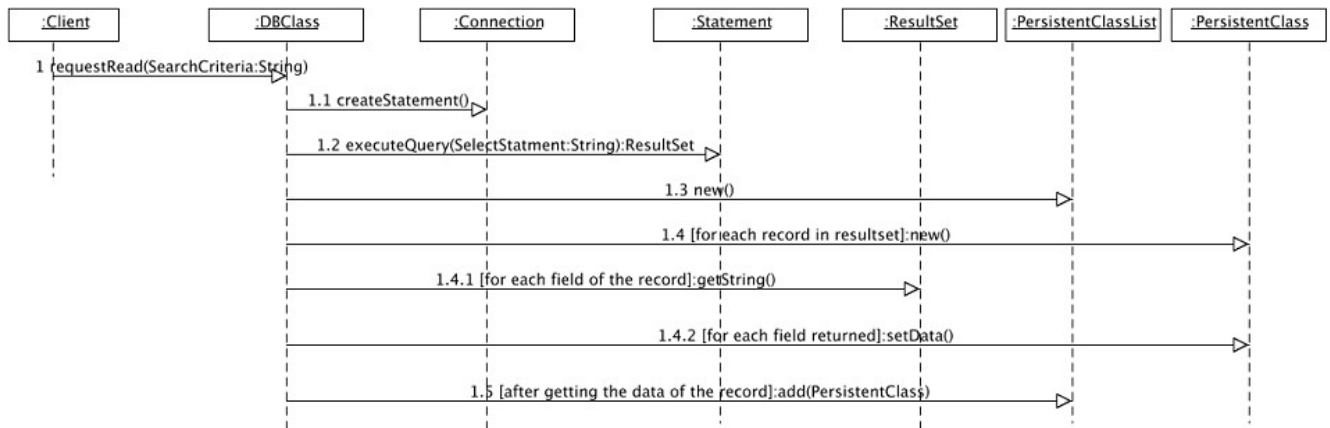


Figura 4: JDBC Read

- A **PersistencyClient** solicita à **DBClass** que recupere registros do banco de dados. A string **SearchCriteria** qualifica que registros serão retornados.
- A **DBClass** cria uma declaração SELECT **Statement** usando o método **createStatement()** de **Connection**.
- A **Statement** é executada via **executeQuery()** e retorna um **ResultSet**.
- A **DBClass** instancia uma **PersistentClassList** para manter os registros que estão no **ResultSet**.
- Para cada registro no **ResultSet**, a **DBClass** instancia uma **PersistentClass**.
- Para cada campo no registro, atribui o valor do campo (**getString()**) ao atributo apropriado na **PersistentClass** (**setData()**).
- Após obter todos os dados do registro e mapeá-los para os atributos da **PersistentClass**, adiciona a **PersistentClass** à **PersistentClassList**.

4. **JDBC Update**. Executa a instrução UPDATE de SQL. Modifica os valores de um registro existente no banco de dados. Assume que uma conexão já esteja estabelecida. O diagrama de sequência é mostrado na Figura 5.

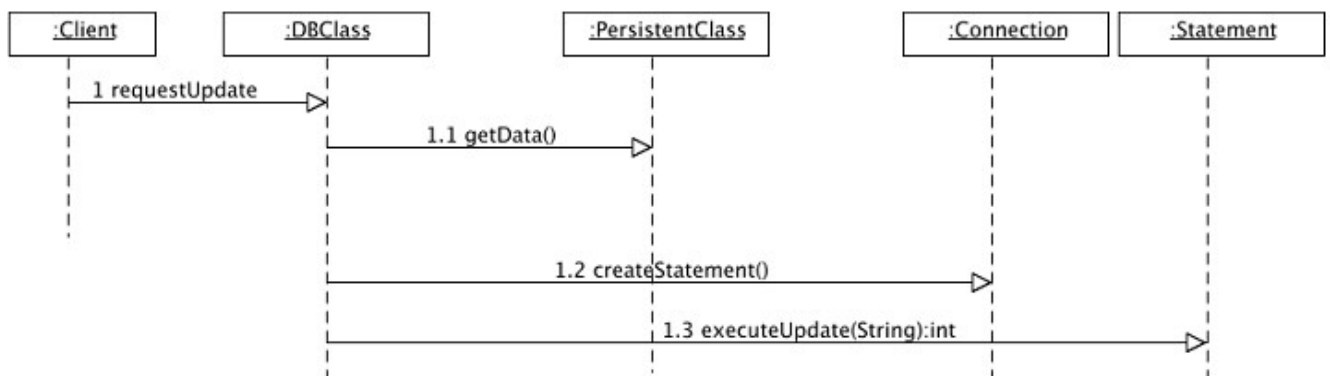


Figura 5: JDBC Update

- **PersistencyClient** solicita à **DBClass** que atualize um registro.
- A **DBClass** recupera o dado apropriado da **PersistentClass**. A **PersistentClass** deve prover uma rotina de acesso para todos os dados persistentes que a **DBClass** precisar. Isto irá garantir acesso externo a certos atributos persistentes que deveriam ser privados. Esta é uma prática para que você mantenha o conhecimento sobre persistência fora da classe que encapsula os atributos persistentes.
- A **Connection** irá criar uma instrução UPDATE.
- Uma vez que a **Statement** é construída, a instrução UPDATE é executada e o banco de dados é atualizado com os novos dados da classe, vale dizer que o estado da classe será atualizado no banco de dados.

5. **JDBC Delete**. Executa uma instrução DELETE em SQL. Esta instrução elimina os registros em um banco de dados. Presume que uma conexão com o banco de dados já esteja estabelecida. O diagrama de seqüência é mostrado na Figura 6.

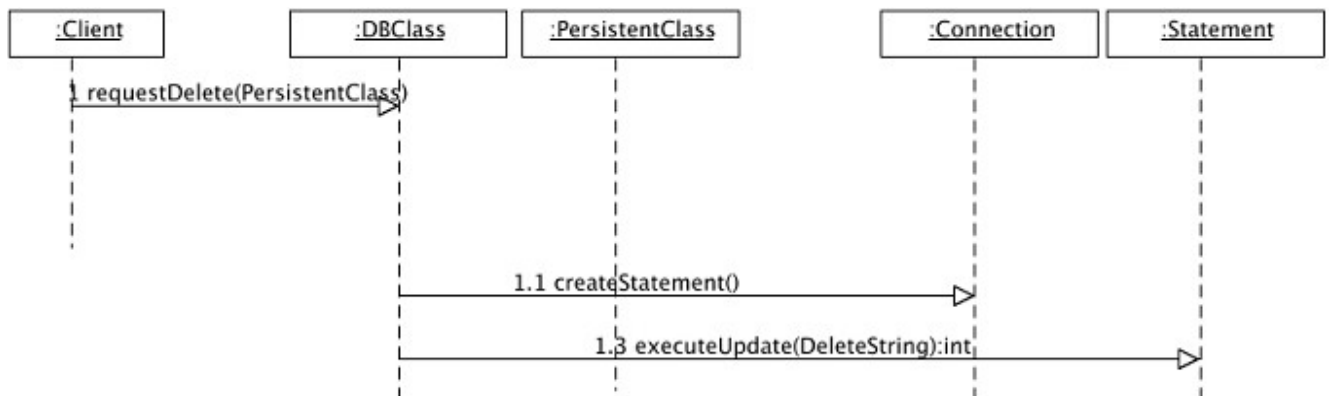


Figura 6: JDBC Delete

- A **PersistencyClient** solicita à **DBClass** que exclua o registro.
- A **DBClass** cria uma instrução DELETE e a executa por meio do método **executeUpdate()**.

3. Implementando JDBC Design Pattern

Esta sessão mostra como escrever código Java que conecta e solicita serviços de um banco de dados, obedecendo o padrão JDBC Design Pattern. Para trabalhar com JavaDB no NetBeans, adicione as bibliotecas `derby.jar`, `derbyclient.jar` e `derbytools.jar` como bibliotecas do projeto. A Figura 7 mostra os arquivos jar com parte do projeto atual (Organicshop).

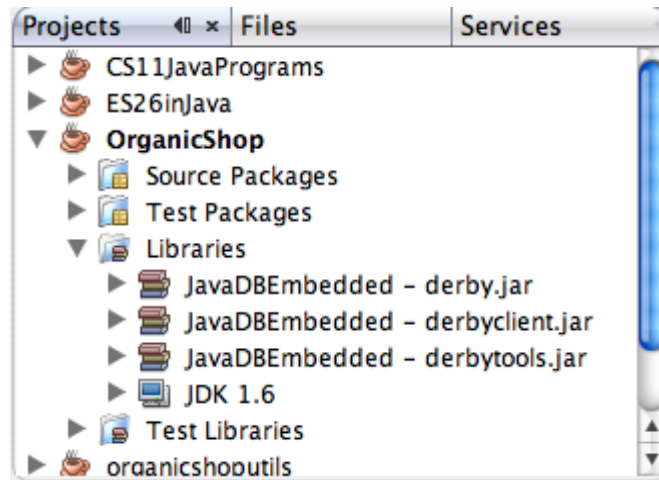


Figura 7: Bibliotecas JavaDB/Derby

O pacote `java.sql` é usado porque contém classes e interfaces para manipulação de bancos de dados relacionais em Java.

```
import java.sql.*;
```

A instrução `import` especifica que queremos usar as classes e interfaces definidas no pacote `java.sql`. A Tabela 2 lista as classes comuns que são usadas pelo programa Java.

O exemplo que será usado é o The Organic Shop. Vamos assumir que uma aplicação Java irá prover o inventário de uma loja em particular. O diagrama de classes está representado pela Figura 8.

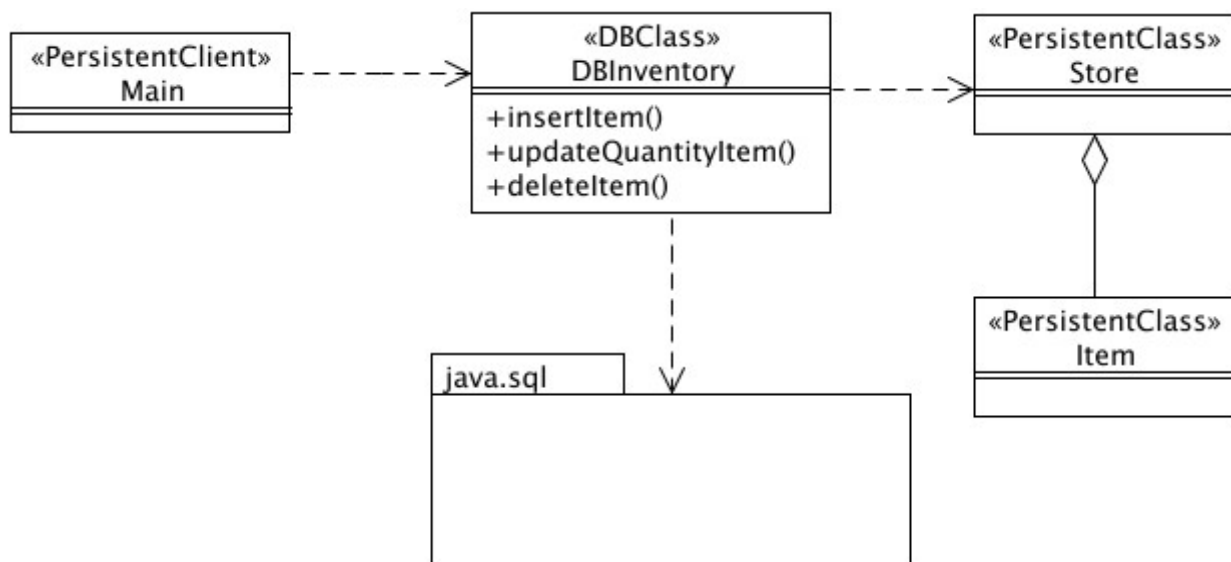


Figura 8: Diagrama de Classe do Inventory By Store Java

A Tabela 3 descreve cada classe do Diagrama de Classes Inventory By Store Java segundo as orientações do padrão JDBC Design Pattern.

Classe	Método	Descrição do Método
Main É o cliente que solicita serviços ao banco de dados. Usa a classe <code>Store</code> como um meio para manter os dados para ler e escrever no banco de dados.		
DBInventory É a DBClass responsável por ler e escrever dados persistentes do inventário no banco de dados. Utiliza a classe <code>Store</code> para ler e escrever dados no banco de dados.	<code>getInventoryByStore()</code>	Retorna um objeto <code>Store</code> que representa a loja e seus itens de inventário.
	<code>saveNewItem()</code>	Salva um novo item vendido pela loja.
	<code>updateCount()</code>	Atualiza a quantidade em estoque de um item em particular.
	<code>deleteAnItem()</code>	Exclui um item vendido pela loja.
	<code>establishConnection()</code>	Método privado que estabelece uma conexão com o banco de dados.
	<code>closeConnection()</code>	Método privado que fecha uma conexão estabelecida com o banco de dados.
Store É a classe persistente que contém o inventário de uma loja. Um de seus atributos é uma lista de itens.	<code>toString()</code>	Retorna uma string que contém o número da loja e o nome com o inventário da loja, i.e., todos os itens com suas respectivas quantidades em estoque e nível de operação.
Item É uma classe persistente que contém o item que é vendido, sua quantidade e nível operacional.	<code>toString()</code>	Retorna uma string que contém o código do item, descrição do item, quantidade em estoque e nível de operação de um item vendido na loja.

Tabela 3: Descrição das Classes de Inventory By Store

3.1. Estabelecendo e Fechando uma Conexão

A classe `Main` é o `Persistent Client` que solicita serviços para o banco de dados. Ele se comunica com a classe `DBInventory`. O código abaixo mostra a declaração de seus atributos e os métodos `establishConnection()` e `closeConnection()`. Observe que os atributos da classe `DBInventory` incluem objetos das classes `Connection`, `Statement` e `ResultSet`.

Antes que possamos solicitar qualquer serviço do banco de dados, uma conexão precisa ser estabelecida. O método `establishConnection()` nos dá o código para conexão com o banco de dados. As linhas seguintes especificam a URL (Uniform Resource Locator) do banco de dados que ajuda o programa a localizar o banco de dados, `username` e `password`. A URL especifica o protocolo para comunicação (**jdbc**), o subprotocolo (**derby**) e o nome do banco de dados (**organicshop**). O `username` e a `password` também são especificados para autenticação no banco de dados; no exemplo, o `username` é "nbuser" e a `password` é "nbuser".

```
String url = "jdbc:derby://localhost:1527/organicshop";
String username = "nbuser";
String password = "nbuser";
```

A classe que define o *driver* para o banco de dados deve estar carregada antes de o programa conectar-se ao banco de dados. As linhas seguintes carregam o *driver*. Neste exemplo, o *driver*

embutido do JavaDB é carregado para permitir que qualquer programa Java tenha acesso a um banco de dados JavaDB.

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

Para estabelecer uma conexão, um objeto do tipo *Connection* é criado. Ele gerencia a conexão entre o programa Java e o banco de dados. Ele também provê suporte para a execução de instruções SQL. No exemplo, a seguinte instrução provê esta conexão.

```
this.theConnection = DriverManager.getConnection(url, username, password);
```

Note que as classes *Class* e *DriverManager* estão dentro de um bloco try-catch. Se um erro ocorrer, uma exceção do tipo *ClassNotFoundException* é disparada para a instrução *Class.forName()* quando o *driver* não puder ser carregado ou localizado, enquanto uma *SQLException* é levantada quando uma conexão não puder ser estabelecida.

```
package organicshop.database.dbclass;

import java.sql.*;
import java.util.List;
import java.util.ArrayList;
import organicshop.database.persistent.Store;
import organicshop.database.persistent.Item;

public class DBInventory {
    private Store theStore;
    private int storeNumber;
    private Connection theConnection;
    private Statement theStatement;
    private ResultSet theResultSet;

    private void establishConnection() {
        String url = "jdbc:derby://localhost:1527/organicshop";
        String username = "nbuser";
        String password = "nbuser";
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            this.theConnection =
                DriverManager.getConnection(url, username, password);
        } catch (ClassNotFoundException cnfx) {
            System.err.println("Organic Shop: Cannot Load Driver");
            cnfx.printStackTrace();
            System.exit(1);
        } catch (SQLException sqlx) {
            System.err.println(
                "Organic Shop: Unable to connection to the database");
            sqlx.printStackTrace();
            System.exit(1);
        }
    }

    private void closeConnection() {
        try {
            System.out.println(
                "Organic Shop: SQL processing done, and closing " +
                "the connection.");
            this.theConnection.close();
        } catch (SQLException sqlx) {
            System.err.println(
                "Organic Shop: Unable to close the connection");
            sqlx.printStackTrace();
            System.exit(1);
        }
    }
}
```

É claro, após usar o banco de dados, devemos fechar a conexão. O método *closeConnection()* faz

isso. Observe que *this.theConnection.close()* necessita estar dentro de um bloco try-catch. Dispara uma *SQLException* quando uma conexão não puder ser fechada.

```
public Store getInventoryByStore(int theStoreNumber){
    this.storeNumber = theStoreNumber;
    this.establishConnection();
    this.populateStoreData();
    this.populateItemData();
    this.closeConnection();
    return this.theStore;
}
private void populateStoreData() {
    String selectStmt = "SELECT name FROM store WHERE number = " +
        this.storeNumber;
    try{
        this.theStatement = this.theConnection.createStatement();
        this.theResultSet = this.theStatement.executeQuery(selectStmt);
        if (this.theResultSet != null){
            this.theStore = new Store();
            this.theStore.setNumber(this.storeNumber);
            if (this.theResultSet.next()){
                this.theStore.setName(this.theResultSet.getString(1));
            } else {
                this.theStore.setName("No name.");
            }
        } else {
            System.out.println("Organic Shop: No records retrieved.");
        }
        this.theStatement.close();
    } catch (SQLException sqlx){
        sqlx.printStackTrace();
    }
}
private void populateItemData() {
    List<Item> theItemList = null;
    String selectStmt = "SELECT item.code, item.description, quantity," +
        "op_level ";
    selectStmt = selectStmt + "FROM item, inventory ";
    selectStmt = selectStmt + "WHERE inventory.item_code = item.code ";
    selectStmt = selectStmt + "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.setQuantity(this.theResultSet.getInt(3));
                theItem.setLevel(this.theResultSet.getInt(4));
                theItemList.add(theItem);
            }
        }
        this.theStore.setItemList(theItemList);
        this.theStatement.close();
    } catch (SQLException sqlx){
        sqlx.printStackTrace();
    }
}
```

3.2. Consultando o Database

Assumiremos que queremos ver o inventário de uma loja em particular. Vamos consultar o banco de dados Organic Shop para obter esta informação. O código mostra-nos o código de como ler ou

recuperar dados de um banco de dados. Usaremos o *theStore* (Objeto Store) como o objeto de transferência de dados, i.e., o objeto que será usado para manter os dados que resultarão da consulta ao banco de dados. Para ler ou recuperar dados do banco de dados, o método *getInventoryByStore()* é invocado. Ele retorna um objeto *Store* que contém informações da loja e o inventário de seus itens vendidos. Ele chama dois outros métodos para realizar a consulta ao banco de dados e popular a classe persistente, o objeto *theStore*. A *string selectStmt* é usada para definir instruções SELECT que especifica que registros serão recuperados do banco de dados. O objeto *theStatement* é usado para consultar o banco de dados e é instanciado pelo método *createStatement()* do objeto *theConnection*. O método *executeQuery()* do objeto *theStatement* é usado para executar a instrução SELECT como especificado na *string selectStmt*. O resultado da consulta é colocado e referenciado em *theResultSet* que é um objeto do tipo *ResultSet*. Ele contém os registros recuperados pelo banco de dados.

O objeto *ResultSet* possui métodos que retornam dados da coluna da tabela. Se o valor da coluna é do tipo VARCHAR, então usamos o método *getString()*. Se o valor da coluna é um inteiro, então usamos o método *getInt()*. No exemplo abaixo, o atributo *name* do objeto *theStore* é atribuído ao valor da primeira coluna da linha atual referenciada em *theResultSet*.

```
this.theStore.setName(this.theResultSet.getString(1));
```

Uma vez que os valores do objeto *theStore* tenham sido atribuídos, são passados de volta a classe que fez a chamada. Neste caso, a classe *Main*. O código em *Main* que instancia um *theInventory* (Objeto do tipo *DBInventory*, e chama o método *getInventoryByStore()*. Ele espera retornar um objeto *Store* referenciado pela variável *theStore*.

```
// The code is written in Main.java
DBInventory theInventory = new DBInventory();
Store theStore;
Item theItem;
theStore = theInventory.getInventoryByStore(10);
if (theStore != null){
    System.out.println(theStore.toString());
} else {
    System.out.println("The Store is null.");
}
```

3.3. Inserindo um Registro

Vamos assumir que a loja decidiu vender um novo item. O inventário deste item deve ser mantido no banco de dados. O código Java que insere um registro no banco de dados. Dois métodos são definidos na classe *DBInventory*; chamados, *saveNewItem()* e *insertItem()*. O primeiro método é usado pelo cliente persistente para invocar a inclusão. Todavia, o segundo método encarrega-se da atual inclusão.

Dentro do método *insertItem()*, a *string insertStmt* é usada para manter a instrução atual INSERT-statement com o conjunto de valores apropriados. O processo de consulta é similar, usamos o objeto *theConnection* para criar um objeto *theStatement*. Ao invés de usar o método *executeQuery()*, usamos o método *executeUpdate()*. Por último, o código é cercado por um bloco try-and-catch para que no caso de falha no processo, estejamos prontos para tratar uma possível exceção do tipo *SQLException* gerada.

```
public void saveNewItem(Item theItem1){
    this.theItem = theItem1;
    this.establishConnection();
    this.insertItem();
    this.closeConnection();
}
private void insertItem(){
    String insertStmt = "INSERT INTO inventory VALUES (";
    insertStmt = insertStmt + this.theItem.getStoreNumber() + ",";
    insertStmt = insertStmt + this.theItem.getCode() + ",";
    insertStmt = insertStmt + this.theItem.getQuantity() + ",";
    insertStmt = insertStmt + this.theItem.getLevel() + ")";
    try{
```

```

        this.theStatement = this.theConnection.createStatement();
        this.theStatement.executeUpdate(insertStmt);
        System.out.println("Organic Shop: Row has been inserted.");
        this.theStatement.close();
    } catch (SQLException sqlx){
        sqlx.printStackTrace();
    }
}

```

O código a seguir dentro de **Main** (o cliente persistente) mostra como chamar o método **saveNewItem()**.

```

DBInventory theInventory = new DBInventory();
Item theItem = new Item();

// Enviar novos valores ao item
theItem.setStoreNumber(10);
theItem.setCode(1002);
theItem.setQuantity(1500);
theItem.setLevel(500);

// Chamar o método saveNewItem enviando theItem para inserir
theInventory.saveNewItem(theItem);
theStore = theInventory.getInventoryByStore(10);
if (theStore != null){
    System.out.println(theStore.toString());
} else {
    System.out.println("The Store is null.");
}

```

3.4. Atualizando um Registro

Vamos supor que precisamos atualizar o valor do atributo quantidade do novo registro que acabamos de inserir. O código abaixo demonstra como atualizar o valor. Dois métodos estão definidos na classe DBInventory; chamados: **updateQuantityItem()** e **setQuantity()**. O segundo método executa uma instrução UPDATE. Note que a estrutura é a mesma do processo de inserção, exceto pela instrução INSERT, agora especificamos uma instrução UPDATE.

A string **updateStmt** declarada no método **setQuantity()** especifica a instrução UPDATE usada para modificar o valor do atributo quantidade.

```

//This code is written in DBInventory.java.
public void updateQuantityItem(Item theItem1){
    this.theItem = theItem1;
    this.establishConnection();
    this.setQuantity();
    this.closeConnection();
}
private void setQuantity() {
    String updateStmt = "UPDATE inventory SET ";
    updateStmt = updateStmt + " quantity = " + this.theItem.getQuantity();
    updateStmt = updateStmt + " WHERE store_no = " +
        this.theItem.getStoreNumber();
    updateStmt = updateStmt + " AND item_code = " + this.theItem.getCode();
    try{
        this.theStatement = this.theConnection.createStatement();
        this.theStatement.executeUpdate(updateStmt);
        System.out.println("Organic Shop: Row has been updated.");
        this.theStatement.close();
    } catch (SQLException sqlx){
        sqlx.printStackTrace();
    }
}

```

O código abaixo é usado para invocação. Da mesma maneira, o código é escrito em Main.java. Note que o atributo quantidade de theItem foi modificado. Então, passamos o theItem para o

método `updateQuantityItem()` para atualização.

```
//The following code is written in Main.java

//Assume that an item has been used before.
//Let us set the new quantity.
theItem.setQuantity(1340);

//Call to update the quantity at hand
theInventory.updateQuantityItem(theItem);

theStore = theInventory.getInventoryByStore(10);
if (theStore != null){
    System.out.println(theStore.toString());
} else {
    System.out.println("The Store is null.");
}
```

3.5. *Eliminando um Registro*

Finalmente, não irá vender quaisquer dos itens inseridos anteriormente. O código abaixo demonstra como eliminar um registro. Note que eles tem estrutura de código similar aos processos de inserção e atualização. O método `removeItem()` executa a remoção. A string `deleteStmt` define a instrução DELETE.

```
// This code is written in DBInventory.java
public void deleteItem(Item theItem1){
    this.theItem = theItem1;
    this.establishConnection();
    this.removeItem();
    this.closeConnection();
}
private void removeItem(){
    String deleteStmt = "DELETE FROM inventory ";
    deleteStmt = deleteStmt + "WHERE store_no = " +
        this.theItem.getStoreNumber();
    deleteStmt = deleteStmt + " AND item_code = " + this.theItem.getCode();
    try{
        this.theStatement = this.theConnection.createStatement();
        this.theStatement.executeUpdate(deleteStmt);
        System.out.println("Organic Shop: Row has been deleted.");
        this.theStatement.close();
    } catch (SQLException sqlx){
        sqlx.printStackTrace();
    }
}
```

A invocação do método `deleteItem()` é mostrada no código abaixo que está escrito em `Main.java`. Estaremos passando o objeto `theItem` para ser excluído.

```
theInventory.deleteItem(theItem);
```

Todos os exemplos usam objetos da classe `Statement`. Existem duas outras classes que podem ser usadas que herdam de `Statement`.

1. A classe `PreparedStatement`. A principal diferença entre `Statement` e `PreparedStatement` é que `PreparedStatement` é pré-compilada. Quando você usa um objeto `Statement` para executar uma instrução SQL, a instrução é enviada direto para o DBMS, onde será compilada. Com `PreparedStatement`, o DBMS poderá simplesmente executar a versão já compilada da instrução SQL.

O objeto `PreparedStatement` também pode ser usado com instruções SQL que não recebem qualquer parâmetro. Embora, na grande maioria das vezes, você irá utilizar instruções SQL que recebem parâmetros. A vantagem de utilizar instruções SQL que recebem parâmetros é que você pode usar a mesma instrução suprindo-a com diferentes valores a cada vez que executá-la. O

exemplo abaixo mostra como usar um objeto PreparedStatement com parâmetros, trata-se de uma modificação do método setQuantity() definido na sessão Atualizando um Registro.

```
private void setQuantity(){
    String updateStmt = "UPDATE inventory SET ";
    updateStmt = updateStmt + " quantity = ?";
    updateStmt = updateStmt + " WHERE store_no = ?";
    updateStmt = updateStmt + " AND item_code = ?";
    PreparedStatement thePreparedStatement;
    try{
        this.thePreparedStatement =
            this.theConnection.createStatement(updateStmt);
        this.thePreparedStatement.setInt(1,theItem.getQuantity());
        this.thePreparedStatement.setInt(2,theItem.getStoreNumber());
        this.thePreparedStatement.setInt(3,theItem.getItemCode());
        this.thePreparedStatement.executeUpdate(updateStmt);
        System.out.println("Organic Shop: Row has been updated.");
        this.thePreparedStatement.close();
    } catch (SQLException sqlx){
        sqlx.printStackTrace();
    }
}
```

Objetos PreparedStatement são criados usando o método createStatement de um objeto Connection object, da mesma maneira que um objeto Statement() é criado. As seguintes instruções Java criam um objeto PreparedStatement chamado thePreparedStatement.

```
String updateStmt = "UPDATE inventory SET ";
updateStmt = updateStmt + " quantity = ?";
updateStmt = updateStmt + " WHERE store_no = ?";
updateStmt = updateStmt + " AND item_code = ?";
this.thePreparedStatement = this.theConnection.createStatement(updateStmt);
```

A string updateStmt representa a instrução UPDATE-statement que modifica o atributo quantidade de um item em particular da loja. O sinal (?) na string o lugar marcado para os parâmetros ou valores passados para a instrução SQL-statement. Os valores serão supridos com a chamada de métodos setXXX apropriados definidos na classe PreparedStatement. Se o valor que você quer substituir por um sinal (?) é um int Java, você chamará um método setInt(). Se o valor que você deseja substituir é uma String Java, você deverá chamar um método setString(), e assim sucessivamente. Em geral, existe um método setXXX() para cada tipo jprimitivo da linguagem Java. No nosso exemplo, as seguintes instruções Java atribuem os valores dos parâmetros.

```
this.thePreparedStatement.setInt(1,theItem.getQuantity());
this.thePreparedStatement.setInt(2,theItem.getStoreNumber());
this.thePreparedStatement.setInt(3,theItem.getItemCode());
```

A primeira instrução Java (thePreparedStatement.setInt(1,theItem.getQuantity())) atribui o valor para o primeiro parâmetro do objeto thePreparedStatement usando o valor retornado por theItem.getQuantity(). O primeiro argumento do método chama setInt(), que é 1, representando a posição do parâmetro em PreparedStatement. O segundo argumento, é o valor da atribuição.

Da mesma forma, a segunda instrução, configura o segundo parâmetro de thePrepareStatement utilizando o valor retornado por theItem.getStoreNumber(). Por último, a terceira instrução substitui o terceiro parâmetro de thePreparedStatement pelo valor retornado por theItem.getItemCode(). Para eecutar a thePreparedStatement, o método executeUpdate() é invocado.

Será melhor usar o objeto PreparedStatement em situações que requeram mais atualizações porque, em termos de performance, ele ajuda o sistema a suprimir a fase de compilação toda vez que uma instrução SQL-statement é executada. As instruções SQL-statement são pré-compiladas.

Note que uma vez que um parâmetro é configurado com um valor, ele matém o valor até que um novo valor lhe seja atribuído, ou o método clearParameters() seja chamado.

A modificação do código para usar a classe `PreparedStatement` no lugar da classe `Statement` fica por sua conta como um exercício deste capítulo.

2. A classe `CallableStatement`. JDBC lhe permite chamar uma stored procedure ou stored function em uma aplicação usando um objeto `CallableStatement`. Assim como `Statement` e `PreparedStatement`, isto é feito com um objeto `Connection` aberto. Um objeto `callableStatement` contém apenas uma chamada para uma stored procedure ou function; ele não contém as próprias storeds procedures ou função itself.

A classe `CallableStatement` é uma subclasse de `PreparedStatement`, logo, um objeto `CallableStatement` pode receber parâmetros assim como um objeto `PreparedStatement`. Além disso, um objeto `CallableStatement` possui parâmetros de saída, ou parâmetros que servem às duas coisas entrada e saída. São os parâmetros `INOUT` (input/output). Estes parâmetros e o método `execute` são usados raramente. Um exemplo com o uso de `CallableStatement` é apresentado na sessão `JDBC Stored Procedures e Functions` nesta lição.

4. Servidor de Banco de Dados-do lado Programação

Aplicações-do lado métodos são métodos invocados do lado da aplicação. **Banco de Dados-do lado das procedures** são métodos invocados no âmbito do Banco de Dados, por exemplo, JavaDB.

Banco de Dados-do lado procedure podem ser os mesmos métodos lado-aplicação. A diferença esta como são invocados. Métodos são escritos uma única vez e onde estes são invocados-aplicações ou internas SQL-instruções- determina se uma aplicação é do lado da ou método do lado do banco de dados.

Esta seção lida com o detalhe com métodos do lado do banco de dados database-side, i.e., métodos que são invocados no banco de dados. Estas invocações definidas em stored procedures ou functions. Nesta seção discutiremos especificamente especiais pra programação para JavaDB.

4.1. JDBC Stored Procedures e Functions

Uma **Stored Procedure** é uma subrotina que é avaliada para acessar o sistema de banco de dados relacional. Esses representam todas operações semelhantes a `enterAnOrder()` ou `createNewCustomer()`. Uma **Stored Function** calcula resultados escalares, e impondo obrigatório domínio. JavaDb implementam o corpo de procedures e functions em métodos Java, i.e., chamandos de métodos Java. A habilidade para escrever functions e procedures na ponte Java completa um atributo da APIs Java no ambiente SQL na lógica do lado do servidor. Uma function ou procedure podem chamar de biblioteca padrão Java, de qualquer padrão de extensão Java, ou outras bibliotecas de terceiros que poderemos mostrar mais tarde nos exemplos.

JavaDB atualmente suporta escrever procedures em linguagem de programação Java, which segue o Padrão SQL. Com estas procedures Java, a implementação da procedure, um método estático público Java na classe Java, é compilado fora do banco de dados, tipicamente arquivo dentro de um arquivo jar e apresenta para o banco de dados com CREATE PROCEDURE-instrução ou CREATE FUNCTION-instrução. Deste modo, o CREATE PROCEDURE-instrução ou CREATE FUNCTION-instrução é conhecida como operação atômica "definir e armazenar". A compilação Java para uma procedure ou function pode armazenar no banco de dados usando o padrão SQL procedure SQLJ.INSTALL_JAR ou pode ser armazenado fora do banco de dados no path de classe da aplicação.

Diferenças entre Procedures e Functions

Estes são uma sobreposição entre stored procedure e functions mas cada uma pode também fazer coisas que a outra não pode. Ale disso, a sintaxe para invocar são diferentes. Um stored function pode ser executar uma parte da instrução SQL semelhante aquela SELECT-clausula ou a WHERE-clausula. Estas podem invocar triggers; todavia, não podem modifica dados de um banco de dados.

Para invocar uma stored procedure no Editor SQL, usamos o CALL-instrução enquanto para invocar uma function, usamos VALUES-instrução. Comparação da stored procedure e functions é listada na Tabela 4.

Descrição	Procedure	Function
Execução numa Triger	✗ (but ✓ in 10.2)	✓
Retorno do resultado efetivado(s)	✓	✗
Processo OUT/INOUT Parâmetros	✓	✗
Executar SQL SELECT	✓	✓
Executar INSERT/UPDATE/DELETE	✓	✗
Executar DDL semelhante a um CREATE e DROP	✓	✗
Executar numa SQL Expression	✗	✓

Tabela 4: Compação de Stored Procedures e Functions

Passos na Criação de Stored Procedure e Functions:

1. Criar os Métodos Java que poderão servir no corpo de stored procedure ou function. Arquivar dentro de um arquivo jar. No código abaixo, temos a definição de uma classe chamada Routines que contém dois métodos, de nomes,, insertNewItem() que insere um novo item na tabela, e countItem() que retorna o número de itens vendidos de um particular estoque. A classe é arquivada com o nome organicshoputils.jar.

Quando executamos a stored procedure, o método usado para definir o corpo da procedure mais necessárias para reusar o conexão corrente para banco de dados. Este tipo de conexão é conhecida como uma conexão aninhada. A maneira de obter uma conexão aninhada é usando a url, jdbc:default:connection que é mostrada na instrução a seguir:

```
Connection theConnection =
    DriverManager.getConnection("jdbc:default:connection","nbuser","nbuser");

package organicshop.proc;

import java.sql.*;

public class Routines {
    public static void insertNewItem(String itemName) throws SQLException {
        //Conexão Aninhada
        Connection theConnection =
            DriverManager.getConnection("jdbc:default:connection",
                "nbuser","nbuser");
        PreparedStatement theStatement = theConnection.prepareStatement(
            "INSERT INTO item VALUES (DEFAULT, ?)");
        theStatement.setString(1, itemName);
        theStatement.execute();
        theStatement.close();
        theConnection.close();
    }
    public static int countItem(int storeNumber) throws SQLException {
        int count = 0;
        //Conexão Aninhada
        Connection theConnection = DriverManager.getConnection(
            "jdbc:default:connection","nbuser","nbuser");
        PreparedStatement theStatement = theConnection.prepareStatement(
            "SELECT COUNT(*) FROM inventory WHERE store_no = ?");
        theStatement.setInt(1, storeNumber);
        theStatement.execute();
        ResultSet theResultSet = theStatement.getResultSet();
        if (theResultSet.next()){
            count = theResultSet.getInt(1);
        }
        theResultSet.close();
        theStatement.close();
        theConnection.close();
        return count;
    }
}
```

2. Armazenar no arquivo jar no banco de dados usando o método sqlj. Na ordem para o NetBeans para trabalhar com semelhantes métodos, o derbytools.jar deve ser parte da biblioteca. Para carregar o arquivo jar para o banco de dados database, chame o método sqlj.install(). Um exemplo é mostrado abaixo:

```
CALL sqlj.install_jar
('/Users/weng/NetBeansProjects/organicshoputils/dist/organicshoputils.jar',
 'APP.OrganicShop', 0);
```

O primeiro argumento é o arquivo jar (usando o caminho absoluto), e o segundo argumento é identificar para o arquivo jar. O nome do identificador segue a convenção do padrão

SQLIdentifier92. A figura a seguir mostra como este é executado no NetBeans.

Os próximos passos para registra o classpath para incluir o arquivo jar. Este é feito através da fixação da propriedade `derby.database.classpath`. Usando o seguinte:

```
CALL SYCS_UTIL.SYCS_SET_DATABASE_PROPERTY(
    'derby.database.classpath', 'APP.OrganicShop');
```

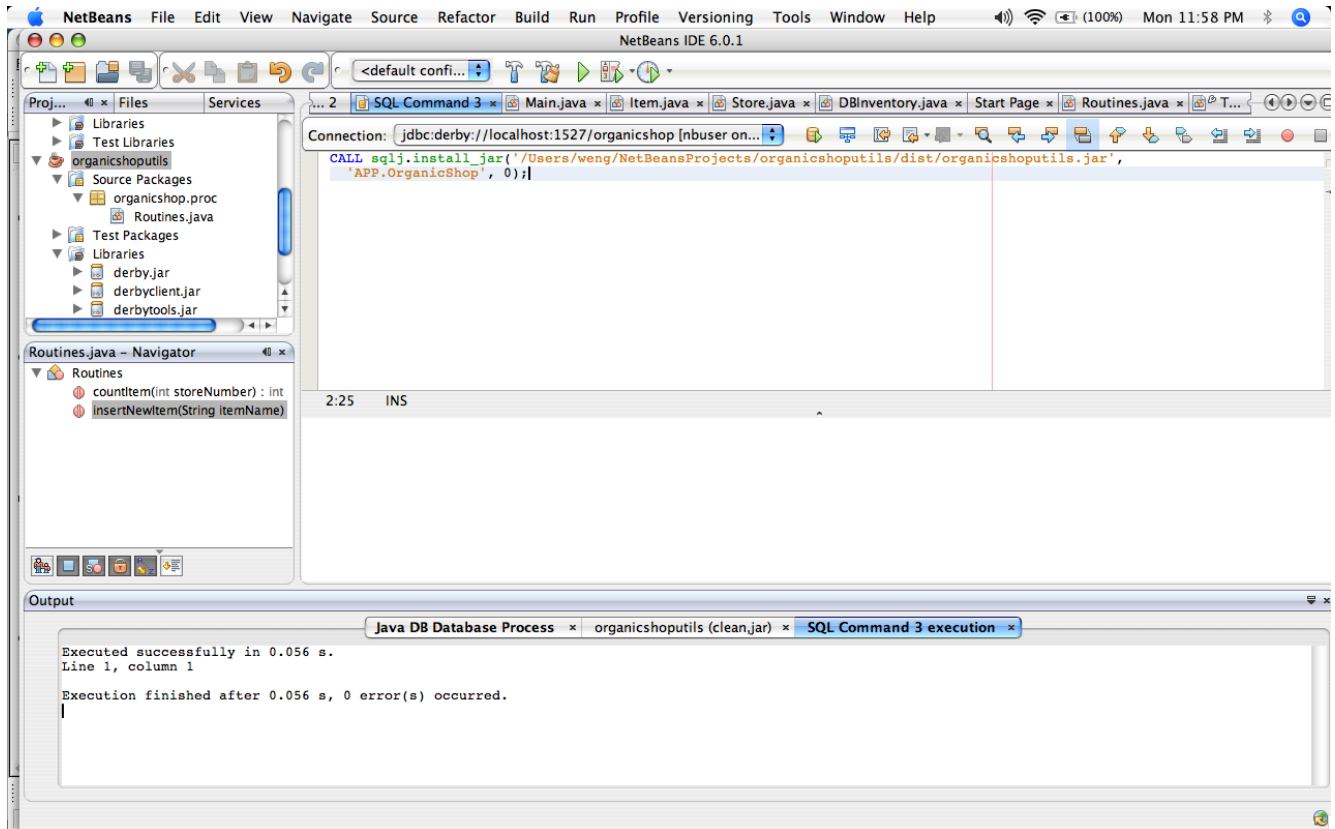


Figura 9: Executando métodos sqlj no NetBeans

3. Crie o procedimento ou função utilizando os comandos SQL *CREATE PROCEDURE* e *CREATE FUNCTION* respectivamente.

O comando *CREATE PROCEDURE* permite a criação de procedimentos armazenados Java, que podem ser chamados utilizando o comando *CALL*. A sintaxe é apresentada na Figura 9.

O nome do procedimento segue as regras do SQL92Identifier. *ProcedureParameter* mostra como definir parâmetros para o procedimento. *IN*, *OUT*, e *INOUT* define o parâmetro. Se não especificado, o valor padrão para um parâmetro é *IN*. O nome do parâmetro deve ser único dentro de um procedimento.

```
CREATE PROCEDURE procedure-Name ( [ ProcedureParameter
    [, ProcedureParameter] ] * ) [ ProcedureElement ] * procedure-Name

ProcedureParameter:
[ { IN | OUT | INOUT } ] [ parameter-Name ] DataType

ProcedureElement:
{
    | [ DYNAMIC ] RESULT SETS INTEGER
    | LANGUAGE { JAVA }
    | EXTERNAL NAME string
    | PARAMETER STYLE JAVA
    | { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA }
}
```

Figura 10: Sintaxe do comando *CREATE PROCEDURE*

Tipos de dados como BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, e XML não são permitidos como parâmetros em um comando CREATE PROCEDURE.

A Tabela 5 lista os elementos do procedimento do comando CREATE PROCEDURE.

Elemento do Procedimento	Descrição
DYNAMIC RESULT SETS integer	Indica o limite superior estimado dos resultados retornados para o procedimento. O padrão é nenhum (zero) resultado dinâmico.
LANGUAGE JAVA	Indica que o gerenciador do banco de dados irá chamar o procedimento como um método estático público em uma classe Java.
EXTERNAL NAME string	Especifica o método Java a ser chamado quando o procedimento é executado, e toma a seguinte forma: nome_da_classe.nome_do_método O Nome Externo não pode ter nenhum espaço entre suas letras.
PARAMETER STYLE JAVA	Isto especifica que o procedimento utilizará uma convenção de passagem de parâmetros em conformidade com as especificações da linguagem Java e com as Rotinas SQL. Parâmetros INOUT e OUT serão passados com o matrizes simples de entrada para facilitar os valores retornados. Os resultados são retornados por parâmetros adicionais para o método Java do tipo java.sql.ResultSet [] que são passados por matrizes simples de entrada.
CONTAINS SQL	Isto indica que os comandos SQL que não lêem nem modificam dados SQL podem ser executados por procedimentos armazenados. Comandos que não são suportados por nenhum procedimento armazenado retornam um erro diferente.
NO SQL	Isto indica que o procedimento armazenado não pode executar nenhum comando SQL.
READS SQL DATA	Isto indica que alguns comandos SQL que não modificam dados podem ser incluídos em um procedimento armazenado. Comandos que não são suportados em nenhum procedimento armazenado retornam um erro diferente.
MODIFIES SQL DATA	Isto indica que um procedimento armazenado pode executar qualquer comando SQL com exceção dos comandos que não são suportados em um procedimento armazenado.

Tabela 5: Elementos do Procedimento

Os elementos de procedimento podem aparecer em qualquer ordem, mas cada tipo de elemento só pode aparecer uma vez. Uma definição de procedimento deve conter estes elementos:

- LANGUAGE
- PARAMETER STYLE
- EXTERNAL NAME

O código a seguir mostra um exemplo de CREATE PROCEDURE para o método insertNewItem() da classe Routines. O nome do procedimento no banco de dados é insertItem();

```
CREATE PROCEDURE insertItem(IN name VARCHAR(50))
PARAMETER STYLE JAVA
LANGUAGE JAVA
EXTERNAL NAME 'organicshop.proc.Routines.insertNewItem'
MODIFIES SQL DATA;
```

Figura 11: Exemplo de comando CREATE PROCEDURE

O comando CREATE FUNCTION cria funções Java, às quais você pode utilizar em uma expressão. A sintaxe é exibida no código abaixo.

```
CREATE FUNCTION function-name ( [ FunctionParameter
[, FunctionParameter] ] * ) RETURNS ReturnDataType [ FunctionElement ] *
function-Name
```

```

FunctionParameter:
[ parameter-Name ] DataType

FunctionElement:
{
| LANGUAGE { JAVA }
| EXTERNAL NAME string
| PARAMETER STYLE ParameterStyle
| { NO SQL | CONTAINS SQL | READS SQL DATA }
| { RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT }
}

```

Os nomes dos parâmetros da função devem ser únicos em uma função. Tipos de dados, tais como, BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, e XML não são permitidos como parâmetros em um comando CREATE FUNCTION. A Tabela 6 mostra a definição dos elementos da função.

Elemento do Procedimento	Descrição
LANGUAGE JAVA	Indica que o gerenciador do banco de dados irá chamar o procedimento como um método estático público em uma classe Java.
EXTERNAL NAME string	Especifica o método Java a ser chamado quando o procedimento é executado, e toma a seguinte forma: nome_da_classe.nome_do_método O Nome Externo não pode ter nenhum espaço entre suas letras.
CONTAINS SQL	Isto indica que os comandos SQL que não lêem nem modificam dados SQL podem ser executados por procedimentos armazenados. Comandos que não são suportados por nenhum procedimento armazenado retornam um erro diferente.
NO SQL	Isto indica que o procedimento armazenado não pode executar nenhum comando SQL.
READS SQL DATA	Isto indica que alguns comandos SQL que não modificam dados podem ser incluídos em um procedimento armazenado. Comandos que não são suportados em nenhum procedimento armazenado retornam um erro diferente.
RETURNS NULL ON NULL INPUT	Isto especifica que a função não é invocada se qualquer dos argumentos de entrada for nulo. O resultado é o valor nulo.
CALLED ON NULL INPUT	Isto especifica que a função é invocada se qualquer ou todos os argumentos de entrada forem nulos. Esta especificação quer dizer que a função deve ser codificada para testar se há valores nulos de argumentos. A função pode retornar um valor nulo ou não-nulo. Esta é a configuração padrão.

Tabela 6: Elementos da Função

Os elementos da função podem aparecer em qualquer ordem, mas cada tipo de elemento só pode aparecer uma vez. Uma definição de função deve conter estes elementos:

- LANGUAGE
- PARAMETER STYLE
- EXTERNAL NAME

O código a seguir mostra um exemplo.

```

CREATE FUNCTION countItemInStore(storeNumber INTEGER) RETURNS INTEGER
PARAMETER STYLE JAVA
LANGUAGE JAVA
READS SQL DATA
EXTERNAL NAME 'organicshop.proc.Routines.countItem';

```

4. Opcionalmente, teste o procedimento armazenado ou função utilizando o comando CALL ou o comando VALUES respectivamente. A sintaxe do comando CALL e VALUES é exibida abaixo

```
CALL procedureName([arguments [,arguments]*]);
```

```
VALUES functionName([arguments [,arguments]*]);
```

Exemplos são encontrados abaixo. O último comando SELECT chama a função armazenada countItemInStore() como parte da instrução SQL.

```
-- inserts new item called Spanish Paprika in the item table
CALL insertItem('Spanish Paprika');
```

```
--returns the number of items sold for store number 30
VALUES countItemInStore(30);
```

```
--use the function in a query
SELECT store_no, count(*)
FROM inventory
GROUP BY store_no
HAVING count(*) <= countItemInStore(10);
```

5. Utilize o procedimento armazenado ou função em um programa de cliente ao utilizar um objeto CallableStatement. O código a seguir mostra a chamada de um procedimento armazenado em um programa de cliente Java.

```
public class TestProcedure {
    public static void main(String args[]) {
        String url = "jdbc:derby://localhost:1527/organicsshop";
        String username = "nbuser";
        String password = "nbuser";
        Connection theConnection;
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            theConnection = DriverManager.getConnection(url,username,password);
            CallableStatement theStatement =
                theConnection.prepareCall("{ CALL insertItem(?) }");
            theStatement.setString(1,"Siling Labuyo");
            theStatement.execute();
            theConnection.close();
        } catch (ClassNotFoundException cnfx){
            System.err.println("Organic Shop: Cannot Load Driver");
            cnfx.printStackTrace();
            System.exit(1);
        } catch (SQLException sqlx){
            System.err.println(
                "Organic Shop: Unable to connection to the database");
            sqlx.printStackTrace();
            System.exit(1);
        }
    }
}
```

Neste exemplo, nós instanciamos um objeto CallableStatement chamado theStatement com um argumento que chama o procedimento armazenado insertItem(). O comando Java é exibido abaixo:

```
CallableStatement theStatement = theConnection.prepareCall(
    {CALL insertItem(?)});
```

O sinal de interrogação (?) encontrado na chamada do procedimento é utilizado para guardar o lugar do argumento que é especificado pelo método setString() do comando, como especificado a seguir:

```
theStatement.setString(1, "Siling Labuyo");
```

O método aceita dois argumentos. O primeiro argumento significa a posição do argumento na chamada do procedimento. O segundo é o valor que está sendo passado. No exemplo acima, 1 significa o primeiro argumento com o valor "Siling Labuyo". Para chamar o procedimento, nos invocamos o método `execute()` do objeto `theStatement`. Como a seguir:

```
theStatement.execute();
```

Outro exemplo é exibido no Error: Reference source not found. Aqui é chamada uma função armazenada.

```
public class TestFunction {
    public static void main(String args[]){
        String url = "jdbc:derby://localhost:1527/organicsshop";
        String username = "nbuser";
        String password = "nbuser";
        Connection theConnection;
        ResultSet theResultSet;
        int count = 0;
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            theConnection = DriverManager.getConnection(url,username,password);
            int storeNumber = 30;
            CallableStatement theStatement =
                theConnection.prepareCall("{VALUES countItemInStore(?)})");
            theStatement.setInt(1, storeNumber);
            theResultSet = theStatement.executeQuery();
            if (theResultSet.next()){
                count = theResultSet.getInt(1);
            } else {
                count = 0;
            }
            theConnection.close();
            System.out.println("Store no. " + storeNumber + " has " + count +
                " items.");
        } catch (ClassNotFoundException cnfx){
            System.err.println("Organic Shop: Cannot Load Driver");
            cnfx.printStackTrace();
            System.exit(1);
        } catch (SQLException sqlx){
            System.err.println(
                "Organic Shop: Unable to connection to the database");
            sqlx.printStackTrace();
            System.exit(1);
        }
    }
}
```

Neste exemplo, instanciamos um `CallableStatement` chamado `theStatement` com um argumento que chama a função armazenada `countItemInStore()`. O comando Java é exibido abaixo:

```
CallableStatement theStatement = theConnection.prepareCall
    ({? = CALL countItemInStore(?)});
```

O sinal de interrogação (?) encontrado na chamada da função é utilizado para guardar o lugar do valor a ser retornado pela função, e do valor do argumento que é passado para a função `countItemInStore()`. Para ligar o sinal à saída ou entrada, utilizamos o seguinte comando Java.

```
theStatement.registerOutParameter(1,Type.INTEGER);
theStatement.setInt(2, storeNumber);
```

O primeiro comando indica que estamos esperando um valor inteiro a ser passado ao programa realizador da chamada, e está ligado ao primeiro sinal de interrogação na chamada da função. O segundo método liga o segundo sinal de interrogação com o argumento que está sendo passado para a função. Neste caso, 2 significa o segundo sinal de interrogação enquanto `storeNumber` é o valor que está sendo passado para a função `countItemInStore()`. Para chamar a função, nós

chamamos o método `executeUpdate()` do `theStatement`. Como a seguir:

```
theStatement.executeUpdate();
```

Para recuperar o valor, é utilizado o método `getInt()` do `theStatement`.

```
count = theStatement.getInt(1);
```

Deletando um Procedimento Armazenado e Funções

O comando `DROP PROCEDURE` é utilizado para deletar ou remover um procedimento do banco de dados. Identifica o procedimento particular a ser removido. A sintaxe é exibida abaixo.

```
DROP PROCEDURE procedure-name
```

O comando `DROP FUNCTION` é utilizado para deletar ou remover uma função no banco de dados. Ele identifica a função particular a ser removida. A sintaxe é exibida abaixo.

```
DROP FUNCTION function-name
```

Exemplos são exibidos logo a seguir:

```
DROP PROCEDURE insertItem;  
DROP FUNCTION countItemInStore;
```

Os comandos `DROP` apenas removem a definição do procedimento ou função no banco de dados. Eles não removem os arquivos jar carregados no banco de dados. Para remover os arquivos jar, é preciso utilizar métodos `sqlj`. O método `remove_jar()` é utilizado para remover o arquivo no banco de dados.

```
CALL sqlj.remove_jar('APP.OrganicShop',0)
```

O comando acima remove o `organicshoputils.jar` que foi carregado como o corpo da estrutura do procedimento ou função.

Outro método importante do `sqlj` que manipula o arquivo jar é o `replace_jar` que realoca o arquivo existente jar no banco de dados. Normalmente, isso é útil se foi modificada parte de nosso código. Isso permite substituir o antigo arquivo jar pelo novo arquivo. Por exemplo:

```
CALL sqlj.replace_jar  
( '/Users/weng/NetBeansProjects/organicshoputils/dist/organicshoputils.jar',  
  'APP.OrganicShop' )
```

5. Exercícios

1. O método `populateStoreData()` e o método `populateItemData()` como exibidos no comando `SELECT` para recuperar registros no banco de dados. Modifique o código de tal forma que o objeto `PreparedStatement` seja utilizado.
2. O método `insertItem()` como exibido no comando `INSERT` para inserir um registro no banco de dados. Modificar o código de tal forma que o objeto `PreparedStatement` seja utilizado.
3. O método `removeItem()` como exibido no comando `INSERT` para inserir um registro no banco de dados. Modificar o código de tal forma que o objeto `PreparedStatement` seja utilizado.
4. Utilizando o JDBC Design Pattern, implemente os códigos Java para banco de dados do problema 1 no capítulo 5 (O gerente do apartamento).
 1. Criar os métodos apropriados que manipulem os registros no banco de dados que incluam:
 - incluir um registro
 - remover um registro
 - modificar as colunas de um registro
 2. Criar os procedimentos armazenados a seguir:
 - `insertRoomType()`, que insere uma nova classificação de quarto
 - `changeRoomStatus()`, que alteram os status do quarto. O status do quarto pode ser VAGO ou OCUPADO.
 3. Criar a função armazenada:
 - retornar o tipo de quarto de um quarto em particular
 - retornar o número de quartos a partir de um tipo de quarto
5. Utilizando o JDBC Design Pattern, implemente os códigos Java para banco de dados desenvolvidos no problema 2 na lição 5 (A loja de Elétrica).
 1. Criar os métodos apropriados que manipulem os registros no banco de dados que incluam:
 - incluir um registro
 - remover um registro
 - modificar de colunas de um registro
 2. Criar os procedimentos armazenados a seguir:
 - `insertClient()`, que insere um registro de cliente
 - `changeApplianceStatus()`, que altera o status do dispositivo para FIXO.
 3. Criar a função armazenada:
 - retornar o status de um dispositivo em particular
 - retornar o número de dispositivos que está sendo consertado por um técnico
6. Utilizando o JDBC Design Pattern, implemente os códigos Java para banco de dados desenvolvidos para o e-Lagyan System
 1. Criar os métodos apropriados que manipulem os registros no banco de dados que incluam:
 - incluir um registro
 - remover um registro

- modificar as colunas de um registro
- 2. Criar os procedimentos armazenados a seguir:
 - insertAccount(), que insere um novo cliente no sistema
- 3. Criar a função armazenada:
 - retornar o balanço atual de um cliente
 - retornar o número de cartões vendidos por um cliente

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.

Banco do Brasil

Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec

Suporte e apoio financeiro e logístico a todo o processo.

Borland

Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB

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

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 Professional 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 Professional 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 Platform 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	Carlos Hilner Ferreira Costa	Kleberth Bezerra Galvão dos Santos
Alberto Ivo da Costa Vieira	Daniel Noto Paiva	Luiz Fernandes de Oliveira Junior
Alexandre Mori	Daniel Wildt	Maria Carolina Ferreira da Silva
Alexis da Rocha Silva	Denis Mitsuo Nakasaki	Maricy Caregnato
Aline Sabbatini da Silva Alves	Fábio Antonio Ferreira	Mauricio da Silva Marinho
Allan Wojcik da Silva	Givailson de Souza Neves	Paulo Oliveira Sampaio Reis
Angelo de Oliveira	Jacqueline Susann Barbosa	Ronie Dotzlaw
Aurélio Soares Neto	Jader de Carvalho Belarmino	Seire Pareja
Bruno da Silva Bonfim	João Vianney Barrozo Costa	Sergio Terzella
Carlos Fernando Gonçalves	José Francisco Baronio da Costa	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 Faria – 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 - quantity_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 executãodor que é o módulo responsável pela implementação de uma estratégia de controle de concorrência em particular. O plano de executãodor também é conhecido como o Gerente de Bloqueio, caso o controle de concorrência seja baseado em bloqueios (*lock-based*). O propósito do plano de executãodor 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.

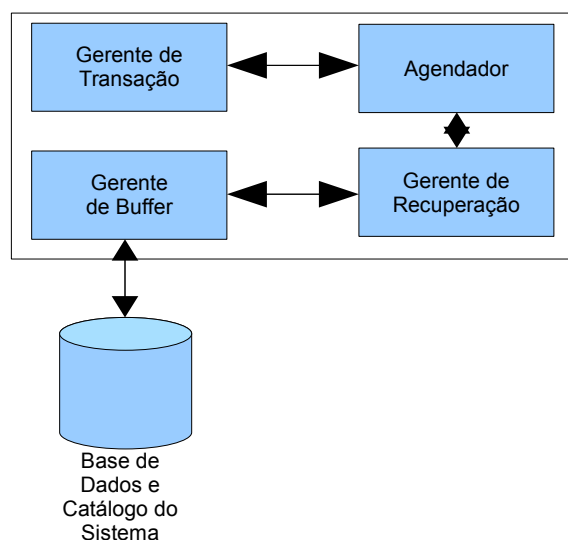


Figura 1: Subsistema de Transações de uma Arquitetura de Base de Dados

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 interferência 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 $quantidade_x$ e $quantidade_z$ 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, \dots, 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; conseqüentemente, 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 _y)	read(qtd _x)
write(qtd _y)	write(qtd _x)
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, $T_i \rightarrow T_j$, se T_j lê o valor de um item escrito por T_i
- uma aresta direcionada, $T_i \rightarrow T_j$, 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

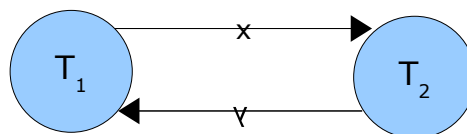


Figura 2: Grafo de Precedência da transação (Duas Transações Concorrentes)

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, \dots, 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 no 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 no plano de execução S_1 , a mesma transação deve executar a escrita final do item de dado x no 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. Executem 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 execuções 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:

T₁	T₂
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 execução 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 qualquer 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 T_2 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 T_1 tenta realizar uma alteração, solicita um bloqueio de escrita em qtd_x . Como foi travado por T_2 , T_1 terá que esperar até que o bloqueio esteja liberado. Isto ocorre somente quando T_2 completou a transação. Quando T_1 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(qtd _x)	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(qtd _x , qtd _z)	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(qtd _z)	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 Inconsistência 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 ela 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 serializável. 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 $T_i \rightarrow T_j$
- 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 ser detectado durante um longo período do tempo.

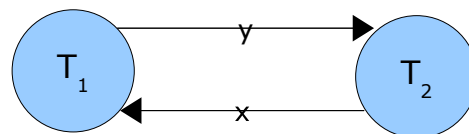


Figura 3: WFG da transação dois em figura 16

Desde que é mais fácil de detectar *deadlocks* e quebrá-los do que preveni-los, a maioria dos sistemas executa o método da detecção 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 prosseguir 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 reiniciadas. 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, $ts(T) \geq write_timestamp(x)$, 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

T₁ possui o *timestamp* ts(T₁)=t₁, T₂ tem ts(T₂)=t₄, e T₃ tem ts(T₃)=t₆ como ts(T₁)<ts(T₂)<ts(T₃).

No tempo t₈, a operação de escrita de T₂ falha sobre o primeiro *timestamp* de escrita, i.e.:

$$ts(T_2) < read_timestamp(qtd_y)$$

$$t_4 < t_7$$

Neste ponto, T₂ realiza um *rollback* e reinicia t₁₄.

No tempo t₁₄, a operação de escrita em qtd_z pode seguramente ignorar porque está sobrescrevendo o que já foi escrito pela transação T₃ no tempo t₇.

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* é 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 possui 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 modificação 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 concorrência permitido. Quanto mais fino o item de dados é, mais a informação é bloqueada 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.

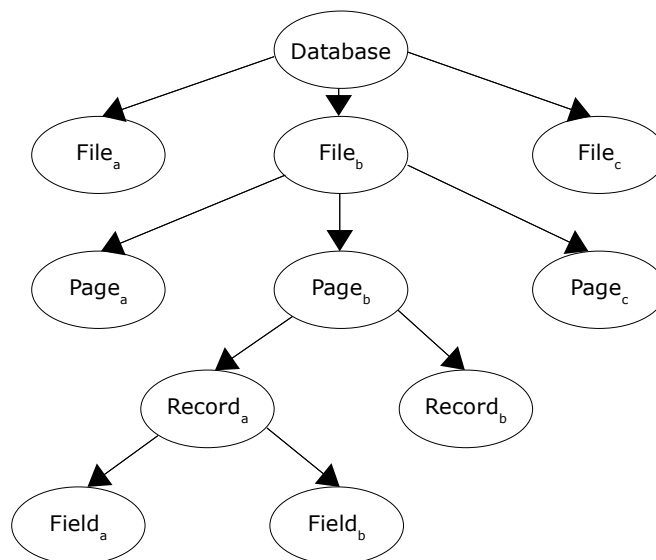


Figura 4: Hierarquia de Granularidade dos Itens de Dados

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. Um bloqueio de intenção compartilhada (IS) conflita com um bloqueio exclusivo.
2. Um bloqueio de intenção exclusivo (IX) conflita 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 serialização 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 precisamos 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 `qtdx`.

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 consequências 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	PPtrs	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	T_1 , T_2				
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 falha 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 informações 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 compartilham 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 concorrência 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).

Código 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.

Código 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:

```

init:
deps-jar:
compile-single:
run-single:
2008-03-20 02:45:01.142 java[514] CLog (0): CFMessagePort: bootstrap_register(): failed 1103 (0x44f), port = 0x15203, name = 'java.ServiceProvider'
See /usr/include/servers/bootstrap_defs.h for the error codes.
2008-03-20 02:45:01.143 java[514] CLog (99): CFMessagePortCreateLocal(): failed to name Mach port (java.ServiceProvider)
Insert Order is successful.
Insert Order Detail successful.
Update of Inventory is successful.
Organic Shop: Transaction is not successful. Rolling back to a consistent state.
-1The statement was aborted because it would have caused a duplicate key value in a unique or primary key constraint or unique index identified by 'SQL
BUILD SUCCESSFUL (total time: 1 minute 10 seconds)

```

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 antes 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 prevêm 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
	<p>Operação A começa.</p> <pre>UPDATE inventory SET qtd = qtd - 50 WHERE store_no = 10 AND item_code = 1001;</pre> <p>Operação B começa.</p> <pre>SELECT * FROM inventory;</pre> <p>Operação B vê os dados atualizados pela operação A mesmo que não tenha encerrado.</p> <p>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:</p> <p>Operação A começa.</p> <pre>SELECT * FROM inventory WHERE store_no = 10 AND item_code = 1001;</pre> <p>Operação B começa.</p> <pre>UPDATE inventory SET qtd = qtd - 50 WHERE store_no = 10 AND item_code = 1001;</pre> <p>Transação B visualiza atualizações pela transação A. Quando operação A emite a mesma declaração, irá produzir um resultado diferente.</p> <p>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:</p> <p>Operação A começa.</p> <pre>SELECT * FROM inventory WHERE qtd < 1000;</pre> <p>Operação B começa.</p> <pre>INSERT INTO inventory VALUES(10, 1004, 1500, 100);</pre> <p>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.</p>
Leitura não repetida	
Leitura fantasma	

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. Para SELECT INTO, FETCH com cursor somente de leitura, seleção utilizada em INSERT, selecção completo / subquery em UPDATE / DELETE, ou selecionar pleno , permite que:
TRANSACTION_READ_UNCOMMITTED	<ul style="list-style-type: none"> 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 isolamento, mais cuidado é necessário para evitar conflitos. Evitando conflitos por vezes significa bloquear as transações. Menor nível de isolamento 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 isolamento 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 compartilhado 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	✓	✓	x
Atualizações	✓	x	x
Exclusivo	x	x	x

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_COMMITTED`, 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_UNCOMMITTED`, 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 feita 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 subsequente 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_COMMITTED, 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 UPDATE 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_UPDATABLE 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 especificá-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 subsequente não pode ter:
 - DISTINCT
 - Aggregates
 - Cláusula GROUP BY
 - Cláusula HAVING
 - Cláusula ORDER BY
- A cláusula FROM na Query subsequente 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_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
theResultSet = theStatement.executeQuery(selectStmt);
theResultSet.moveToInsertRow();
theResultSet.updateString("NAME", "GangStore in Philadelphia");
theResultSet.updateString("ADDRESS", "Philadelphia");
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.updateRow();
```

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_COMMITTED. 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.

Deteção 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 seguintes 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.

Módulo 9

Banco de Dados



Lição 8

Processando Consultas

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

1. Objetivos

Nesta lição, iremos concentrar no tópico de processamento de consulta. A primeira seção fornece uma visão geral de processamento de consulta e suas quatro fases centrais, são elas, decomposição da consulta, otimização da consulta, geração de código e tempo de execução. A segunda seção discute a decomposição da consulta que inclui checagem de sintaxe e semântica da consulta, e transforma linguagem de consulta de alto nível (SQL) para linguagem de baixo nível (implementando expressões algébricas de relacionamento). A Terceira seção discute otimização de consulta. Duas principais são abordadas, especificamente, abordagem heurística e abordagem de estimativa de custo, finalmente, será discutida a técnica *Pipelining* que pode ser usada para melhorar ainda mais o processamento de consultas.

Ao final desta lição, o estudante será capaz de:

- Como realizar consultas
- Decomposição e otimização de consultas
- Técnica *Pipelining*

2. Introdução

O processamento da consulta envolve as atividades que recuperam dados a partir de um banco de dados. Um dos seus objetivos é transformar uma consulta escrita em linguagem de alto-nível, tipicamente SQL, em uma estratégia correta e eficiente executada em linguagem de baixo-nível, e para executar a estratégia para recuperar os dados solicitados.

Como um exemplo, suponha que nos precisamos encontrar, "Todos os itens vendidos em Alabama". A consulta é mostrada no código abaixo:

```
SELECT * FROM inventory, store
WHERE inventory.store_no = store.number
AND (store.address LIKE "%Alabama%");
```

Duas tabelas são usadas nesta consulta, são elas, inventory e store, suponha que existam 50 registros encontrados na tabela store, cada loja vende em media 1000 itens. A tabela do inventário poderá ter mais de 50.000 registros.

Para fazer uma simples ilustração, suponha que não existam índices ou chaves de classificação, também suponha que algum resultado intermediário esta armazenado no disco. Todas as vezes que nos acessarmos o dado, nos encontraremos um registro a cada vez. (Embora na realidade, acesso a disco sejam baseados em blocos que normalmente contem mais que uma linha.)

A álgebra relacional equivalente de consultas para esta Sentença SQL esta como abaixo, e os acessos a disco são calculados da seguinte maneira:

1ª Forma

```
 $\sigma(\text{store.address like "%Alabama\%"} \wedge (\text{inventory.store\_no}=\text{store.number})) (\text{Inventory X Store})$ 
```

Esta consulta calcula o produto cartesiano de inventário e armazenar as tabelas. Isso exigiria (50,000+50) acesso ao disco, e isso cria um relacionamento com (50.000 * 50) linhas. Então essa relação para é lida novamente para testar o predicado da seleção. (50.000 * 50) acessos ao disco são exigidos. O total do custo é calculado do seguinte modo:

$$(50,000+50) + (50,000*50) = 2,550,050 \text{ acessos ao disco}$$

2ª Forma

```
 $\sigma(\text{store.address like "%Alabama\%"})(\text{Inventory} \bowtie \text{Inventory.store\_no}=\text{store.number Store})$ 
```

Esta consulta calcula primeiro a junção das tabelas inventory e store. Isso pode exigir (50.000 + 50) acesso ao disco. Isso cria um relacionamento com (50* 1000) pois sabemos que a loja vende em media 1000 itens. Próximo, isso poderia exigir (50 * 1000) acesso ao disco para fazer a seleção predicado. O total do custo é calculado do seguinte modo:

$$(50.000 + 50) + (50 * 1000) = 100.050 \text{ acessos ao disco}$$

3ª Forma

```
 $\text{Inventory} \bowtie \text{inventory.store\_no}=\text{store.number} \sigma \text{store.address like "%Alabama\%"}$ 
```

Esta consulta calcula primeiro a seleção predicado na tabela store recuperando os registros em Alabama. Isso pode exigir 50 acessos ao disco. Isso cria um relacionamento com 1 linha (Assumindo que somente um registro pode ser encontrado no estado). Então, o resultado ira se juntado com a tabela inventory, requerendo 1000+ 1 acesso ao disco. O total é calculado como mostrado abaixo:

$$50 + 1000 + 1 = 1.051 \text{ acesso ao disco}$$

Obviamente, a terceira consulta fornece a melhor opção e exigiria menos acessos ao disco de forma a obter o registro solicitado.

Processamento de consulta pode ser dividido em quatro fases principais: decomposição da consulta, otimização da consulta, geração de código, execução. Eles são mostrados na figura 2.

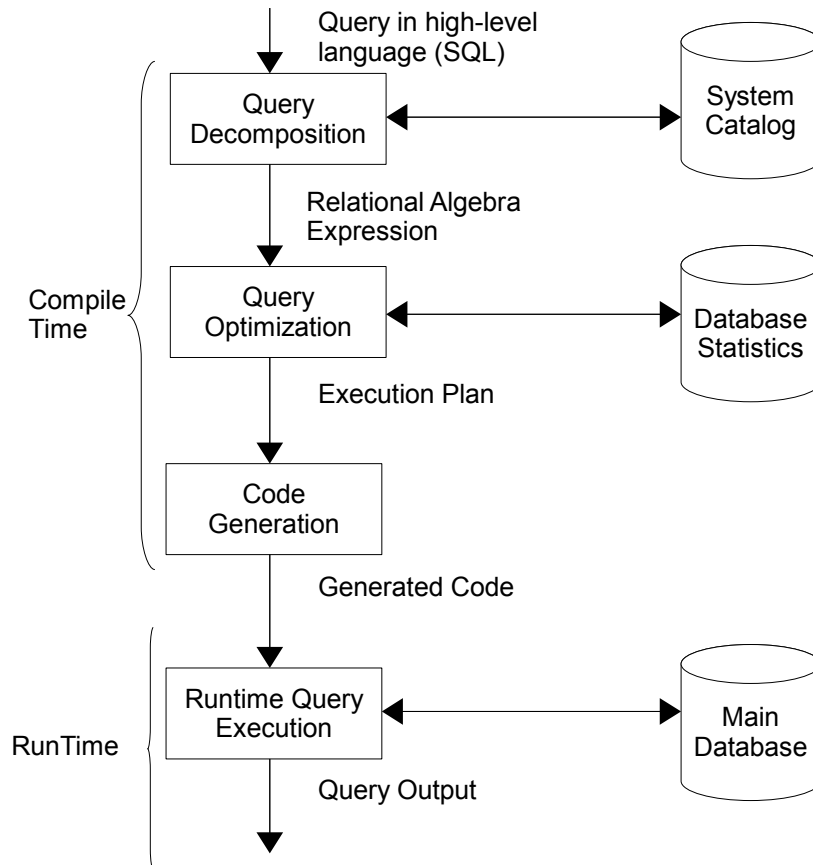


Figura 1: Fases da Consulta

Quatro fases centrais do processamento de consulta:

1. **Decomposição de Consulta.** Esta é a fase em que uma consulta de alto nível é convertido ou transformado em uma consulta de álgebra relacional. É também nesta fase em que é feita uma verificação para assegurar que a consulta é sinteticamente e semanticamente correta. Esta fase utiliza o sistema de catálogo. O sistema de catálogo contém informações sobre os metadados do banco de dados e objetos definidos no dicionário que o banco de dados, tais como tabelas, índices, views e etc. Informações Metadata sobre instâncias de dados são necessárias para o bom funcionamento de qualquer sistema de banco de dados. A sistema de banco de dados usa esta informação quando está atendendo uma solicitação do usuário, quer sob a forma de DML sentenças ou chamadas aos utilitários do banco de dados.
2. **Otimização de consulta.** Esta é a fase que envolve a escolha de uma execução eficiente estratégia de transformação da consulta. Ele utiliza banco de dados estatísticos para ajudar na avaliação sobre as diferentes opções disponíveis na execução de uma consulta. Abrange informações, tais como as relações, atributos, e índices. Também pode incluir cardinalidade das relações, do número de valores distintos para cada atributos, o número de níveis em uma indexação multi-nível etc Todas essas informações são armazenadas como parte do sistema de catálogo e atualizado periodicamente pelo DBMS.
3. **Geração de código.** Esta fase que transforma a estratégia de execução em operações de baixo-nível que podem ser executadas nesta fase.
4. **Consultas em tempo de execução.** Esta fase executa as operações de baixo-nível para recuperar dados do banco de dados. Este usa o banco central de dados onde os dados atuais estão armazenados e recuperados.

Iremos nos concentrar nas fases de decomposição de consulta e otimização de consulta.

3. Decomposição da Consulta

Esta é a primeira fase quando do processamento da consulta. Destina-se a transformar a linguagem de alto-nível, tais como SQL, em uma consulta de álgebra relacional, e também a verificação de erros de sintaxe e semântica. Os estágios normalmente utilizados para a decomposição de consulta são: análise normalização, análise de semântica, simplificação e reconstrução de consulta.

3.1. Análise

Nesta fase, a consulta é analisada por erros de sintaxe. Além disso, verifica que a relação e os atributos especificados na consulta são definidas no sistema de catálogo. Ele também verifica que qualquer operação aplicada ao dicionário objetos são o tipo de objeto adequado. Como um exemplo, considere a consulta mostrada no código a seguir:

```
SELECT itemCode FROM inventory WHERE store_no = '30';
```

Esta consulta será rejeitada pelas seguintes razões:

- Na Cláusula SELECT, a coluna itemCode não é um objeto dicionário válido para a tabela inventory deste que esta não é uma coluna da tabela. Este deveria ser item_code.
- Na cláusula WHERE, a comparação não é compatível desde que store_no é um dado do tipo numérico.

A saída deste estágio pode ser uma árvore de álgebra relacional. Isto é um árvore de consulta que representa o alto-nível consulta em uma representação interna que é mais adequado para processamento. Este é construído como as seguir.

- Um folha do *node* é criada para cada relacionamento de base na consulta.
- Nenhuma folha de *node* é criada para cada relacionamento intermediário produzido por uma operação de álgebra relacional.
- A raiz da árvore representa o resultado de uma consulta.
- A sequência de operações é direcionada para o deixar a raiz

Na Figura 2 vemos um exemplo de uma árvore de álgebra relacional para a seguinte consulta:

```
SELECT * FROM inventory, store
WHERE inventory.store_no = store.number
AND (store.address LIKE "%Alabama%");
```

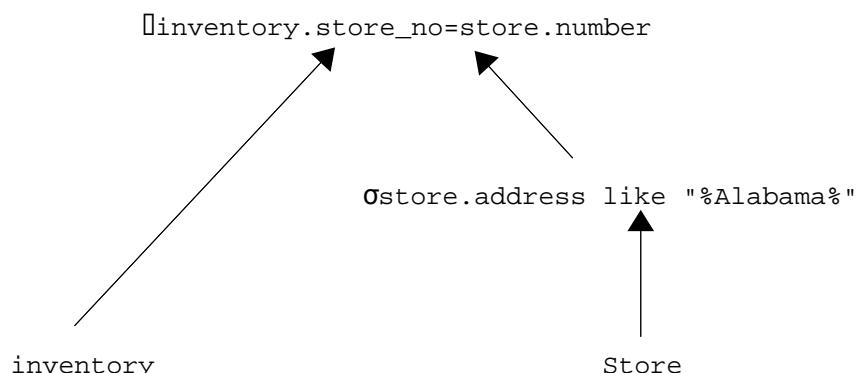


Figura 2: Árvore relacional Algébrica

3.2. Normalização

Nesta fase, a consulta é convertida em uma forma normalizada que pode ser facilmente

manipulada. O predicado (normalmente, uma cláusula WHERE e uma sentença SELECT) que pode ser complexa, pode ser convertida em uma ou duas formas pela aplicação das seguintes regras de transformação.

- **Forma Normal Conjuntiva.** É uma seqüência de conjuntos que esta conectada com o operador \wedge (AND). Cada conjunto contém um ou mais termos conectado pelo operador \vee (OR). Uma seleção conjuntiva contém somente aquelas linhas que satisfazem toda conjunção; i.e., Todas condições conectados pelo operador \wedge é verdade.
- **Forma Normal Disjuntiva.** É uma seqüência disjuntiva que esta conectada pelo operador \vee (OR). Cada disjunção contém um ou mais termos conectados pelo operador \wedge (AND). A seleção disjuntiva contém aquelas linhas formadas pela união de todas as linhas que satisfaça a disjunção.

Considere a consulta mostrada a seguir:

```
SELECT employee.number, lastname, firstname
FROM employee, store
WHERE employee.store = store.number
AND store.number = 10
OR store.number = 30
```

Esta consulta retorna todos os trabalhadores empregados, seja na loja do Alabama (Loja Número 10) ou na loja de Dakota do Norte (Loja Número 30).

Possui a seguinte **Forma Normal Conjuntiva** da cláusula WHERE:

```
(employee.store = store.number)  $\wedge$  (store.number = 10  $\vee$  store.number = 30)
```

E a seguinte **Forma Normal Disjuntiva**:

```
((employee.store = store.number)  $\wedge$  (store.number = 10))  $\vee$ 
((employee.store = store.number)  $\wedge$  (store.number = 30))
```

A Forma Normal Disjuntiva normalmente deixa para replicação predicado. Observe no código que o predicado `employee.store = store.number` é repetido.

3.3. Análise Semântica

É necessária para rejeitar normalizada da consulta que estão incorreta mente formuladas ou são contraditórias. A consulta que não está propriamente formulada pode ter componentes que não contribuirão para a geração do resultado. Normalmente, isso acontece quando há componentes perdidos na consulta. Uma consulta que é contraditória pode ter predicados que não satisfaçam nenhuma linha na tabela.

Para um sub-ajuste das consultas que não têm nenhuma disjunção e negação, as seguintes checagem podem ser realizadas para determinar sua correção

Construindo gráfico de conexão relacional como foi formulado por Wong e Youssefi em 1976. Se o gráfico construído não esta conectado, nos dizemos que a consulta esta corretamente formulada. Para construir o gráfico, use os passos a seguir:

1. Criar um *node* para cada tabela participante da consulta.
2. Criar um *node* para representar o resultado da consulta.
3. Criar uma aresta para cada junção.
4. Criar uma aresta para cada código de operação de projeto.

Exemplo, considere a consulta mostrada no código a seguir:

```
SELECT item.description FROM store, inventory, item
WHERE store.number = inventory.store_no
AND (inventory.store_no = 10 or inventory.store_no = 30);
```

Esta consulta recupera os itens vendido para armazenar números 10 e 30. O Gráfico de conexão

relacional de consulta esta mostrada na figura a seguir. Note que o gráfico não está conectado. Neste caso, esquecemos de juntar *inventory* e *item*. portanto, a consulta está incorretamente formulada.

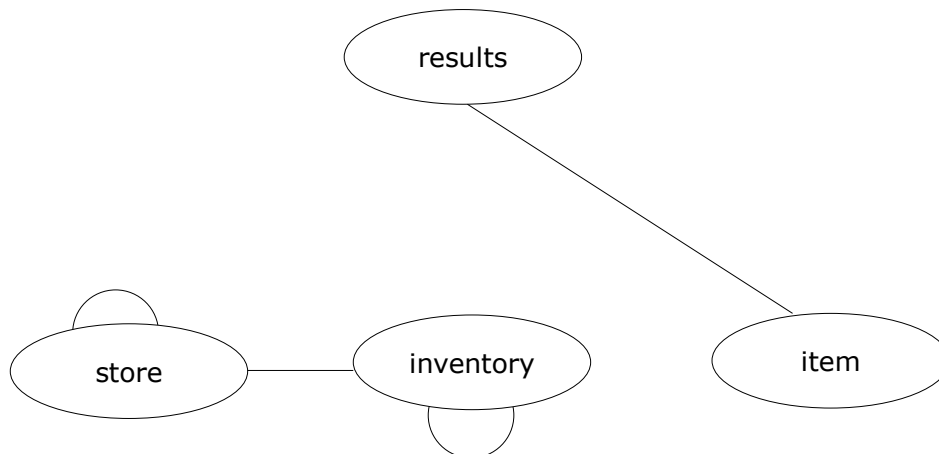


Figura 3: Gráfico de Conexão Relacional

Construindo atributo normalizado de conexão gráficas como foi proposto por Rosendranz e Hunt em 1980. Se o gráfico construído for um círculo para que o valor da soma seja negativo, A consulta é contraditória. Para construir o gráfico, use os seguintes passos:

1. Criar um *node* para cada referência para um atributo e para cada constante 0.
2. Criar uma aresta entre *nodes* que representam uma junção.
3. Criar uma aresta entre um *node* atributo e uma constante de 0 que representa a operação de seleção.
4. O peso das margens $a \rightarrow b$ terá o valor c , se isso representar a condição de desigualdade.
5. O peso das margens $0 \rightarrow a$ terá o valor $-c$, se isso representar uma condição de desigualdade.

Como um exemplo, considere o código abaixo:

```
SELECT store.name, item.description FROM store, inventory, item
WHERE inventory.quantity > 500
AND store.number = inventory.store_no
AND inventory.quantity < 200
AND inventory.item_code = item.code
```

O Gráfico de Conexão do Atributo Normalizado da consulta é mostrado na figura a seguir:

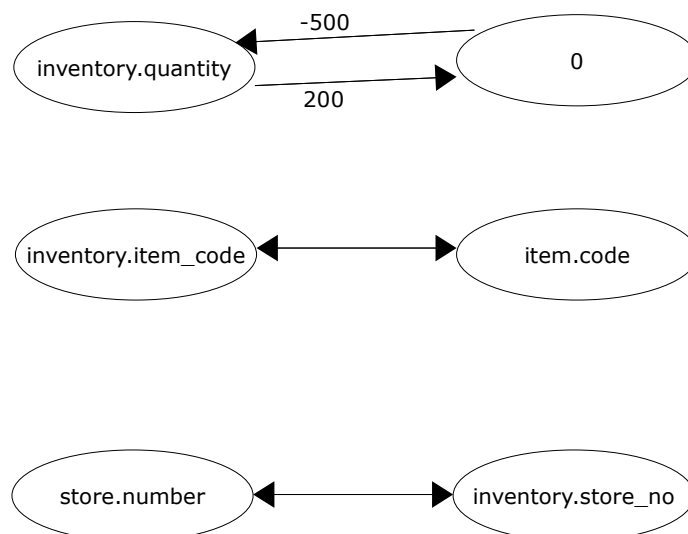


Figura 4: Gráfico de Conexão do Atributo normalizado

Possui um ciclo (inventory.quantity com constantes 0) e uma avaliação negativa tem uma soma (-500 +200 = -300). Isto indica que a consulta é contraditória. Claramente, um item não pode ter um valor para um (quantidade > 500) e (valor < 200), ao mesmo tempo.

3.4. Simplificação

O propósito da fase de simplificação é:

- Detectar qualificações de redundância
- Remover sub-expressões comuns.
- Transformar a consulta em uma semanticamente equivalente, mas de forma mais fácil e eficiente

Neste estágio, restrições de acesso, definições de visualização, e restrições de integridade são considerados. Se o usuário não tem o nível apropriado de privilegio para todos os componentes da consulta, a consulta é rejeitada. Uma inicial e simples otimização é aplicada usando o bom conhecimento das regras de álgebra booleana com estas:

```
p ∧ (p) ≡ p
p ∨ (p) ≡ p
p ∧ false = false
p ∨ false ≡ p
p ∧ true = p
p ∨ true ≡ true
p ∧ (~p) = false
p ∨ (~p) ≡ true
p ∧ (p ∨ q) = p
p ∨ (p ∧ q) ≡ p
```

3.5. Restruturação da Consulta

A consulta é reestruturada para fornecer de forma uma implementação mais eficiente. Reestruturação será discutida em detalhes na próximo seção.

4. Otimização da Consulta: Aproximação Heurística

A Aproximação Heurística otimiza as consultas usando a regra de transformação para converter uma expressão de álgebra relacional em uma expressão equivalente e conhecida por ser mais eficiente. Como mostrado para o código abaixo:

```
SELECT * FROM inventory, store
WHERE inventory.store_no = store.number
AND (store.address LIKE "%Alabama%");
```

É melhor executar a operação de seleção primeiro antes de uma operação JOIN. Nesta seção, veremos algumas regras de transformação que são válidas e apresentam um conjunto de heurística que é conhecido por produzir boas estratégias de execução; embora, não necessariamente podem ser ótimas.

4.1. Regras de Transformação para Operações Algébricas Relacionais

Otimizadores de consulta podem aplicar regras de transformação que permite converter uma expressão de álgebra relacional em uma expressão de álgebra relacional equivalente que é conhecida por ser mais eficiente. As regras serão aplicadas à árvore de álgebra relacional gerada durante a decomposição de consulta. As regras usam as seguintes estruturas:

1. Três relações descritas como R, S e T;
2. R possui atributos $A = \{A_1, A_2, \dots, A_n\}$;
3. S possui atributos $B = \{B_1, B_2, \dots, B_n\}$;
4. p, q e r são predicatos; e
5. L, L1, L2, M, M1, M2 e N representam um conjunto de atributos.

As regras de transformação são:

1. Cascata de Seleção

A operação conjuntiva de seleção pode ser feita em cascata com as operações de seleção individuais (e vice-versa).

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

Esta transformação é conhecida como Cascata de Seleção. Por exemplo:

```
 $\sigma_{\text{inventory.store\_no} = \text{store.number} \wedge \text{item\_code} = 1001}(\text{Inventory}) =$ 
 $\sigma_{\text{inventory.store\_no} = \text{store.number}}(\sigma_{\text{item\_code} = 1001}(\text{Inventory}))$ 
```

2. Seleção Comutativa

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

Por exemplo:

```
 $\sigma_{\text{inventory.store\_no} = \text{store.number}}(\sigma_{\text{item\_code} = 1001}(\text{Inventory})) =$ 
 $\sigma_{\text{item\_code} = 1001}(\sigma_{\text{inventory.store\_no} = \text{store.number}}(\text{Inventory}))$ 
```

3. Sequência de Operações de Projeção

Somente a última na sequência é requerida.

$$\pi_L \pi_M \dots \pi_N(R) = \pi_L(R)$$

Por exemplo:

$$\pi_{\text{number}} \pi_{\text{name}}(\text{Store}) = \pi_{\text{number}}(\text{Store})$$

4. Comutabilidade de seleção e projeção.

Se o predicado **p** envolve somente os atributos na lista de projeção, então a seleção são as

operações de projeção e comutação.

$$\pi_{A_1, \dots, A_m}(\sigma_p(R)) = \sigma_p(\pi_{A_1, \dots, A_m}(R)) \text{ where } p \in \{A_1, A_2, \dots, A_m\}$$

Por exemplo:

$$\pi_{\text{number}, \text{name}}(\sigma_{\text{number}=30}(R)) = \sigma_{\text{number}=30}(\pi_{\text{number}, \text{name}}(R))$$

5. Comutabilidade de *theta-join* (e Produto Cartesiano). Como o *equi-join* e união natural é casos especiais do *theta-join*, esta regra se aplica a ambos.

$$\begin{aligned} R \bowtie_p S &= S \bowtie_p R \\ R \times S &= S \times R \end{aligned}$$

Por exemplo:

$$\begin{aligned} \text{Employee} \bowtie_{\text{employee.store=store.number}} \text{Store} &= \text{Store} \bowtie_{\text{employee.store=store.number}} \text{Employee} \end{aligned}$$

6. Comutatividade de seleção e *theta-join* (e Produto Cartesiano). Se o predicado da seleção, **p**, usa somente atributos de uma das relações que são unidas, então a seleção e a união (ou Produto Cartesiano) das operações de comutação.

$$\begin{aligned} \sigma_p(R \bowtie_r S) &= (\sigma_p(R)) \bowtie_r S \\ \sigma_p(R \times S) &= (\sigma_p(R)) \times S \\ \text{quando } p &\in \{A_1, A_2, \dots, A_m\} \end{aligned}$$

Se o predicado de seleção é um predicado conjuntivo que tem a forma (**p** **q**), onde **p** usa os atributos de **R**, e **q** usa os atributos de **S**, então a seleção é o *theta-join* das operações de comutação.

$$\begin{aligned} \sigma_{p \wedge q}(R \bowtie_r S) &= (\sigma_p(R)) \bowtie_r (\sigma_q(S)) \\ \sigma_{p \wedge q}(R \times S) &= (\sigma_p(R)) \times (\sigma_q(S)) \end{aligned}$$

Por exemplo:

$$\begin{aligned} \sigma_{\text{hiredate} > '01-01-2000' \wedge \text{store.number}=30} &= \\ (\text{Employee} \bowtie_{\text{employee.store=store.number}} \text{Store}) &= \\ (\sigma_{\text{hiredate} > '01-01-2000'}(\text{Employee})) \bowtie_{\text{employee.store=store.number}} &= \\ \sigma_{\text{store.number}=30}(S) \end{aligned}$$

7. Comutatividade de projeção e *theta-join* (ou Produto Cartesiano). Se a lista de projeção é **L** = **L1** \cup **L2** onde **L1** usa atributos de **R**, e **L2** usa atributos de **S**, e, a condição *join* contém atributo em **L**, a projeção e o *theta-join* das operações de comutação.

$$\pi_{L_1 \cup L_2}(R \bowtie_r S) = \pi_{L_1}(R) \bowtie_r \pi_{L_2}(S)$$

Se a condição *join* contém atributos adicionais não achados em **L**, i.e., **M** = **M1** \cup **M2** onde **M1** são atributos achados em **R** e **M2** é atributos achados em **S**, então a operação de projeto final é:

$$\pi_{L_1 \cup L_2}(R \bowtie_r S) = \pi_{L_1 \cup L_2}((\pi_{L_1 \cup M_1}(R)) \bowtie_r (\pi_{L_2 \cup M_2}(S)))$$

Exemplo da primeira regra:

$$\begin{aligned} \pi_{\text{employee.lastname, store.number}} &= \\ (\text{Employee} \bowtie_{\text{employee.store=store.number}} \text{Store}) &= \\ \pi_{\text{employee.lastname}(\text{Employee}) \bowtie_{\text{employee.store=store.number}} \pi_{\text{store.number}} &= \\ \pi_{\text{store.number}}(\text{Store}) \end{aligned}$$

Exemplo da segunda regra:

$$\begin{aligned} \pi_{\text{employee.lastname, store.name}} &= \\ (\text{Employee} \bowtie_{\text{employee.store=store.number}} \text{Store}) &= \\ \pi_{\text{employee.lastname, employee.store}(\text{Employee}) \bowtie_{\text{employee.store=store.number}} &= \\ \pi_{\text{store.name, store.number}}(\text{Store}) \end{aligned}$$

8. Comutatividade de união e interseção (sem diferença fixa).

$$\begin{aligned} R \cup S &= S \cup R \\ R \cap S &= S \cap R \end{aligned}$$

9. Comutatividade de seleção e conjunto de operações (união, intersecção e diferença fixa).

$$\begin{aligned} \sigma_p(R \cup S) &= \sigma_p(R) \cup \sigma_p(S) \\ \sigma_p(R \cap S) &= \sigma_p(R) \cap \sigma_p(S) \\ \sigma_p(R - S) &= \sigma_p(R) - \sigma_p(S) \end{aligned}$$

10. Comutatividade de projeção e união.

$$\Pi_L(R \cup S) = \Pi_L(R) \cup \Pi_L(S)$$

11. Associatividade de *theta-join* (e Produto Cartesiano).

$$\begin{aligned} (R \bowtie S) \bowtie T &= R \bowtie (S \bowtie T) \\ (R \times S) \times T &= R \times (S \times T) \end{aligned}$$

Quando a condição *join*, **q**, usa somente atributos da relação **S** e **T**, a união é dita **associativa** da seguinte maneira:

$$(R \bowtie_p S) \bowtie_q T = R \bowtie_{p \wedge q} (S \bowtie_q T)$$

Por exemplo:

```
(Store ⋈ store.number = inventory.store_no Inventory) ⋈
  inventory.item_code = item.code ∧ store.number = 10 Item =
  Store store.number=inventory.store_no ∧ store.number=10 ⋈
  (Inventory ⋈ inventory.item_code=item.code Item)
```

12. Associatividade de união e intersecção (sem diferenças fixas).

$$\begin{aligned} (R \cup S) \cup T &= R \cup (S \cup T) \\ (R \cap S) \cap T &= R \cap (S \cap T) \end{aligned}$$

4.2. Estratégias de processamento heurístico

Muitos sistemas de bancos de dados relacionais usam heurística para determinar as estratégias que processamento de consulta. Esta seção fornece algumas boas heurísticas que possam ser aplicadas durante o processamento de consulta.

1. Realizar operações de seleção o mais cedo possível. A seleção reduz a cardinalidade da relação. Reduz o processamento subsequente das relações. Seria sábio usar a 1ª regra para cascatear o processamento das relações. Use as regras 2, 4, 6 e 9 para mover a operação de seleção o máximo possível para baixo da árvore. Mantenha os predicados da mesma relação juntos.
2. Combinar o Produto Cartesiano com a operação de seleção subsequente de onde o predicado representa uma condição de junção para uma operação de junção. Reescreva todos os Produtos Cartesianos com um predicado que defina a operação de junção.
3. Usar associação de operações binárias para rearranjar o *node* folha de modo que os *nodes* folhas com as operações de seleção mais restritivas sejam executadas primeiro. Realize a máxima redução (operação de seleção) quanto for possível antes de realizar uma operação binária (tipo uma junção). As regras 11 e 12 referem-se a associação de união e interseção para reordenar operações que resultem na menor junção sendo realizada primeiramente, o que significa que a segunda junção será baseada no primeiro menor operador.
4. Realizar projeção o mais cedo possível. Projeção reduz a cardinalidade da relação, e reduz o subsequente processamento da relação. Use a regra 3 para candidatar as operações de projeção. Use as regras 4, 7 e 10 para mover as operações de projeção o máximo possível para baixo da árvore. Mantenha atributos da mesma relação juntos.
5. Compute expressões comuns uma vez. Se uma expressão comum aparecer mais que uma vez na árvore, e o resultado não for muito grande, guarde o resultado depois de ter

sido computado, e depois reuse.

Um exemplo de como otimização de consulta usando heurística reduz trabalho usando SELECT no código a seguir:

```
SELECT item.description, item.quantity
FROM store, inventory, item
WHERE store.number = inventory.store_no
AND inventory.item_code = item.code
AND store.number = 10;
```

A álgebra relacional para a consulta é,

$$\Pi_{\text{item.description, item.quantity}}(\sigma_{\text{store.number=inventory.store_no} \wedge \text{inventory.item_code=item.code} \wedge \text{store.number=10}}(\text{store} \times \text{inventory}) \times \text{item})$$

A árvore de álgebra relacional é mostrada na figura a seguir:

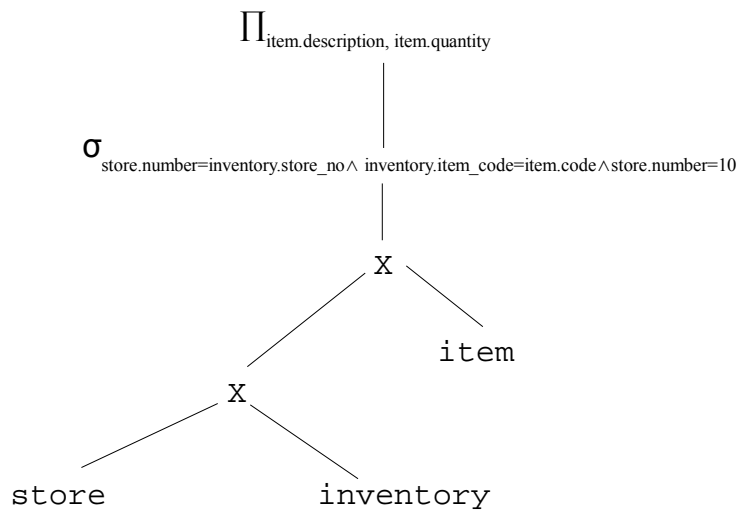


Figura 5: Grupo Simples da Álgebra Relacional

Aplicar a **regra 1** que separa a conjunção da operação de seleção em operações de seleção individuais. Depois, aplicar as **regras 2 e 6** para reordenar os operações de seleção e comutar as operações de seleção e Produto Cartesiano. A árvore resultante é mostrada a seguir:

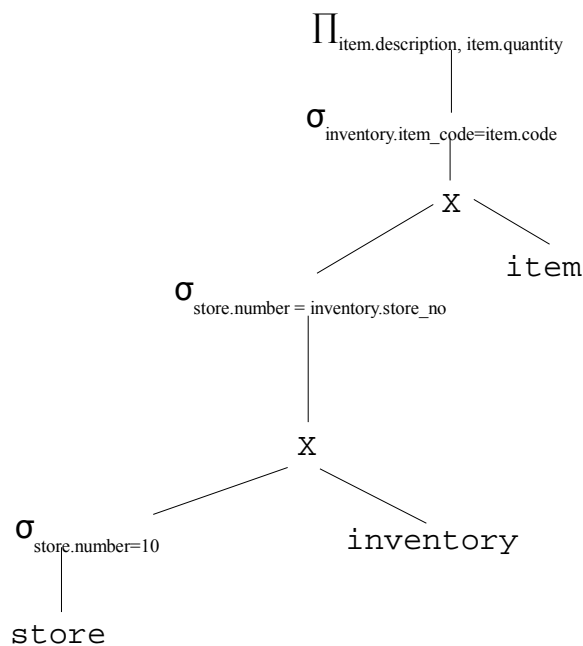


Figura 6: Aplicando as regras 1, 2 e 6

Mudando os Produtos Cartesianos para *equi-joins*, a árvore resultante é mostrada a seguir:

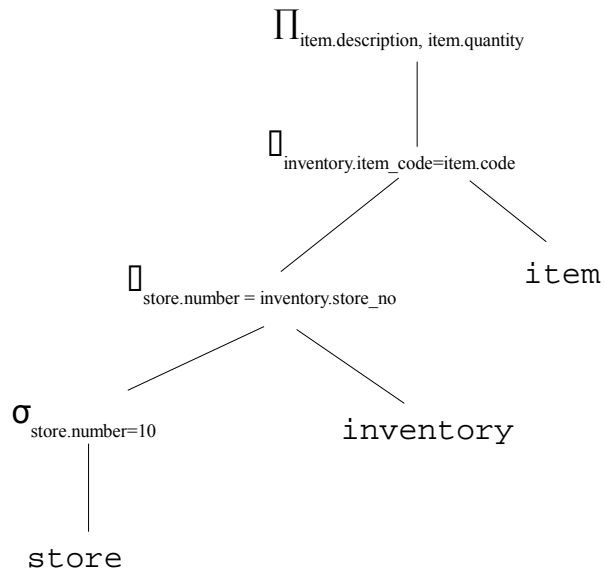


Figura 7: Mudando o produto cartesiano para *equi-joins*

Aplicando as **regras 4 e 7** para mover a projeção para baixo das *equi-joins*, e criando novas projeções conforme requerido. A árvore da álgebra relacional resultante é mostrada a seguir:

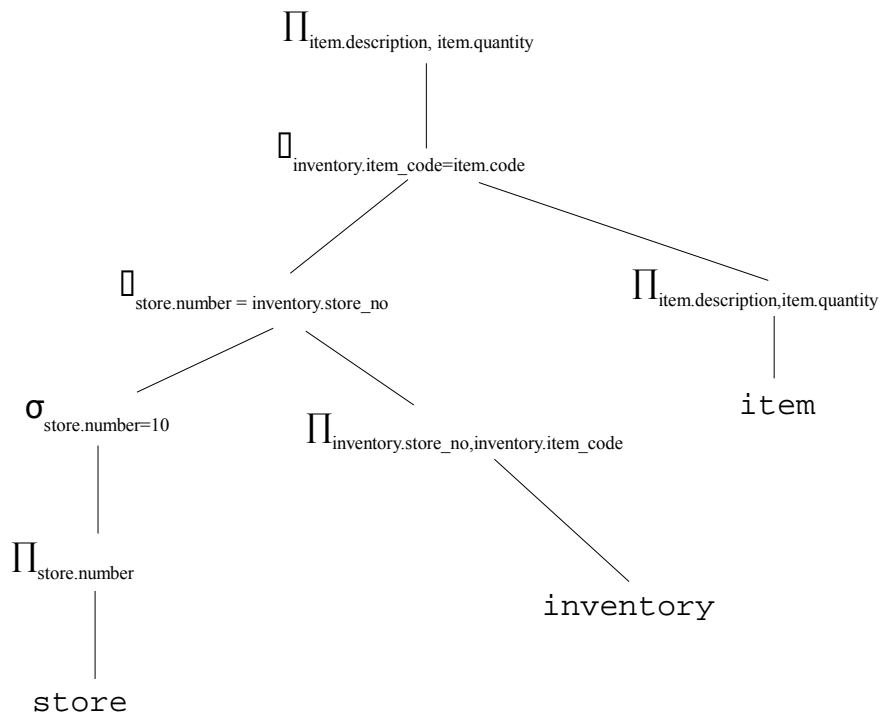


Figura 8: Modelo final da Árvore de Álgebra Relacional

5. Otimização de Consultas: Estimativa de custo para operações de álgebra relacional

O alvo da otimização de consulta é escolher o modelo mais eficiente. Para fazer isso, a implementação do sistema de banco de dados relacional usa fórmulas que estimam o custo para um número de opções, e seleciona a que possui o menor custo. Essa seção examina as diferentes opções disponíveis para implementação nas principais álgebras relacionais. Consideraremos o seguinte:

- Concentrar o custo envolve acessos aos disco desde que eles sejam mais lentos comparados aos acessos à memória.
- Cada estimativa representa o número de acessos requeridos aos blocos de disco.
- Excluir o custo de escrever o resultado no disco.

Como será mostrado, a maioria das estimativas de custo são baseadas na cardinalidade da relação. Precisamos estar habilitados a estimar o custo de relações intermediárias.

Estatísticas de banco de dados são requeridas para estimar o custo de acessos ao disco. Normalmente, um sistema de banco de dados teria estatísticas do banco de dados armazenado em seus catálogos de sistema. O sucesso de estimar o tamanho e custo da consulta vai depender o montante e frequência da informação estatística que os DBMs possuem. Gerlamente, DBMs armazenam o seguinte em seus catálogos de sistema:

Para cada relação R:

- $nTuples(R)$. Esse é o número de linhas em uma relação, ou seja, sua cardinalidade.
- $bFactor(R)$. Esse é o fator de bloco de R, ou seja, o número de linhas que cabe em um bloco.
- $nBlocks(R)$. Esse é o número de bloco requeridos para armazenar R.

Se as linhas de R forem armazenadas juntas fisicamente, então:

$$nBlocks(R) = nTuples(R) / nFactor(R)$$

para cada atributo A:

- $nDistinctA(R)$. Esse é o número de valores distintos que aparecem para o atributo A em relação a R.
- $minA(R)$, $maxA(R)$. Esse são os mínimo e máximo valores possíveis para o atributo A em R.
- $SCA(R)$. Esse é a seleção cardinal do atributo A. Essa é a média de tuplas que satisfazem uma condição equivalente no atributo A.

Se os valores de A são uniformemente distribuidos em R, e existe pelo menos um valor que satisfaz a condição, então:

$$SCA(R) = \begin{cases} 1 & \text{se A é um atributo chave} \\ [nTuples(R) / nFactor(R)] & \text{de outra maneira} \\ [nTuple(R) * ((maxA(R) - c) / (maxA(R) - minA(R)))] & \text{para desigualdade } (A > c) \\ [nTuple(R) * (c - (maxA(R)) / (maxA(R) - minA(R)))] & \text{para desigualdade } (A < c) \\ [nTuple(R) / nDistinctA(R) * n] & \text{para } (A \text{ em } c_1, c_2, \dots, c_n) \\ SCA(R) * SCB(R) & \text{para } (A \wedge B) \\ SCA(R) + SCB(R) - SCA(R) * SCB(R) & \text{para } (A \vee B) \end{cases}$$

Para cada index I multi nível no atributo A:

- $nLevelsA(I)$. Esse é o número de níveis no index I.
- $nlfBlocksA(I)$. Esse é o número de blocos folha em I.

Em geral, as estatísticas são atualizadas periodicamente quando o sistema não está em seus períodos de pico ou quando o sistema está desocupado.

5.1. Operação de Seleção

Há uma série de maneiras de implementar a operação de seleção em função da estrutura do arquivo em que a relação é armazenada, e em qualquer atributo(s) envolvido no predicado que tiver sido indexado ou particionado. As principais estratégias e os seus correspondentes custos de computação é apresentado na Tabela 1.

Estratégias de Operação de Seleção	Custo
Busca Linear (Arquivo desordenado, sem índice)	Para igualdade de condição no atributo chave: $nBlocks(R) / 2$ Caso contrário: $nBlocks(R)$
Busca Binária (Arquivo ordenado, sem índice)	Para igualdade de condição no atributo ordenado: $\log_2(nBlocks(R))$ Caso contrário: $\log_2(nBlocks(R)) + SCA(R) / bFactor(R) - 1$
Igualdade na chave <i>hash</i>	Assumindo sem <i>overflow</i> : 1
Igualdade de condição na chave primária	$nLevels_A(I) + 1$
Desigualdade de condição na chave primária	$nLevels_A(I) + [nBlocks(R) / 2]$
Igualdade de condição ao gerar índice (secundário)	$nLevels_A(I) + [SCA(R) / bFactor(R)]$
Igualdade de condição ao desmontar índice (secundário)	$nLevels_A(I) + SCA(R)$
Desigualdade de condição no secundário B+-tree índice	$nLevels_A(I) + [nlfBlocks_A(I) / 2 + nTuples(R) / 2]$

Tabela 1: Resumo das Estratégias de Estimativa de Custo para as Operações de Seleção

5.2. Operação de Junção

A operação que deu maior trabalho foi a operação de junção, que além de ser um produto cartesiano, é a que mais consome tempo para processar, e temos que garantir que seja o mais eficiente possível. A Tabela 2 mostra as estratégias empregadas para a operação de junção, e o cálculo da estimativa dos custos de transformação.

Estratégias de Operação de Seleção	Custo
União de Laços Aninhados em Bloco	<p>Para um buffer que tem apenas um bloco de R e S: $nBlocks(R) + (nBlocks(R) * nBlocks(S))$</p> <p>Para (nBuffer – 2) blocos para R: $nBlocks(R) + [nBlocks(S) * (nBlocks(R) / (nBuffer - 2))]$</p> <p>Para todos os blocos de R que pode ser lido no buffer do banco de dados: $nBlocks(R) + nBlocks(S)$</p>
União de Laço Aninhados Indexado	<p>Depende do método de indexação: Para uma união do atributo A com uma chave primária: $nBlocks(R) + nTuples(R) * (nLevels_A(I) + 1)$</p> <p>Para a agregação índice I a um atributo A: $nBlocks(R) + nTuples(R) * (nLevels_A(I) + SC_A(R) / bFactor(R))$</p>
União Sort-merge	<p>Para classificar: $nBlocks(R) * [\log_2(nBlocks(R)) + nBlocks(S) * [\log_2(nBlocks(S))]]$</p> <p>Para fundir: $nBlocks(R) + nBlocks(S)$</p>
União de Partes	<p>Para reter índice hash em memória: $3(nBlocks(R) + nBlocks(S)) + 2 * max_partitions$</p> <p>Caso contrário: $2(nBlocks(R) + nBlocks(S)) * [\log_{nBuffer-1}(nBlocks(S)) - 1] + nBlocks(R) + nBlocks(S)$</p>

Tabela 2: Resumo das Estratégias de Estimativa de Custo para as Operações de União

5.3. Operação de Projeção

Por definição, operação de projeção é um operador unário que define a relação, S, que contém um subconjunto de uma relação vertical, R, extraíndo valores dos atributos especificados, e eliminando valores duplicados. Dois passos são necessários para implementar uma operação projeção.

1. Remover todos os atributos que não são requeridos; e
2. Eliminar todas as linhas duplicadas.

Para determinar o custo estimado da projeção, precisamos determinar o custo da operação para cada passo.

1. Determinando o custo estimado para o passo 1 (cardinalidade da operação de projeção). A tabela abaixo mostra o cálculo do custo estimado.

Passo 1 (Cardinalidade da Operação de Projeção)	Custo
Para uma projeção que contém o atributo chave, não é necessário, para a performance, eliminar as duplicidades desde que não exista duplicidade para o resultado da relação.	$nTuples(S) = nTuples(R)$
Para uma projeção que contém um único atributo não-chave: $S = \Pi A(R)$	$nTuples(S) = SCA(R)$
Para uma relação produzida por um projeto cartesiano (que é irrealista)	$nTuples(S) \leq \min \left(nTuples(R) \right)$ M $\prod_{i=1} (nDistinct\ ai(R))$

Tabela 3: Resumo do custo estimado da Etapa 1 da Operação de Projeção

2. Determinar o custo da eliminação das linhas duplicadas. Existem basicamente duas estratégias para este passo que são mostradas na tabela abaixo.

Passo 2 Eliminando Registros Duplicados	Custo
Usando o método de classificação. O objetivo desta passagem é classificar os registros de relações reduzidas, com efeito de arrumar fileiras duplicadas adjacentes umas com as outras, o que torna mais fácil remoção.	<p>Ler os registros de R e copiar para uma relação temporária:</p> $nBlocks(R)$ <p>Classificando os registros:</p> $nBlocks(R) * [\log_2(nBlocks(R))]$ <p>O Custo total é:</p> $nBlocks(R) + nBlocks(R) * [\log_2(nBlocks(R))]$
<p>Usando o método Hashing. Este método é muito utilizando quando um grande número de blocos de buffer estão disponíveis. Abaixo seguem os passos envolvidos:</p> <ol style="list-style-type: none"> 1. Um bloco é atribuída à leitura da relação R, e (nBuffer-1) buffers para a saída. 2. Para cada registro em R, remover atributos não desejados, e aplicar uma função hash, h, com a combinação dos restantes atributos. Escreva a redução na fila para o valor hash. 3. Leia cada um dos (nBuffer-1), na posição da partição. 4. Aplique um segundo valor hash, h2, e inserir o valor hash computado para um hash em memória da tabela hash. 5. Se a linha dividir-se no mesmo valor, verifique se os dois são os mesmos, e elimine o novo se for uma duplicação. 6. Uma vez que uma partição tenha sido processada, escreva os resultados em arquivo de retorno. 	<p>Se o numero de blocos requeridos pela tabela temporaria que resulta do projeto R antes a eliminação da duplicidade é nb, o custo estimado calculado é:</p> $nBlocks(R) + nb$ <p>Isso assume que não existe sobrecarga de hasing.</p>

Tabela 4: Resumo do Custo Estimado para a Eliminação de Duplicidade

5.4. Álgebra relacional conjunto operações

O conjunto de operações binárias, interseção e diferença se aplicam apenas para relações que são

união-compatível, ou seja, relações que tenham estruturas idênticas. Estas operações são normalmente implementadas como segue:

1. Classifique as relações ambas pelos mesmos atributos.
2. Examine através das relações ordenadas depois de obter os seguintes resultados:

Para uma União ($R \cup S$), o resultado é colocado em fila se ele não aparece em nenhuma das relações originais. Eliminar as duplicidades, quando necessário.

Para uma Intersecção ($R \cap S$), o resultado é colocado em fila se ele aparecer nas duas relações originais.

Para a Diferença ($R - S$), é colocada em uma fila o resultado se ele aparece em R mas não em S.

Um algoritmo de junção classificar-fundir pode ser utilizado para todos os casos. O custo estimado para cálculo é:

$$\begin{aligned} & nBlocks(R) + nBlocks(S) + \\ & nBlocks(R) * [\log_2(nBlocks(R))] + nBlocks(S) * [\log_2(nBlocks(S))] \end{aligned}$$

Porque duplicidades são eliminadas quando realizar a união operação, é geralmente difícil de estimar a cardinalidade da operação, mas nós podemos computar para os limites superiores e inferiores, que é:

$$\max(nTuples(R), nTuples(S)) \leq nTuples(T) \leq nTuples(R) + nTuples(S)$$

Para definir a operação de diferença, os limites superiores e inferiores são definidos como:

$$0 \leq nTuples(T) \leq nTuples(R)$$

6. Pipelining¹

Um conceito que permite a melhoria no desempenho de consultas é *pipelining*. Também é conhecido como *on-the-fly processing*. Quando a consulta é processada, são assumidos os resultados de intermediárias operações de álgebra relacional para escrever no disco, i.e., é armazenado em uma relação temporária para processar à próxima operação. Isto é chamado materialização. Uma aproximação alternativa é o *pipeline* dos resultados, i.e., passar os resultados de uma operação para outra operação sem criar uma relação temporária. Isto economiza o custo de criar relações temporárias e reler os resultados. Considere a operação de seleção com um predicado especificado no código abaixo:

```
σitem_code=1001σstore_no=10(inventory)
```

Que pode ser reescrito para:

```
σitem_code=1001σstore_no=10(inventory)
```

Usando uma técnica *non-pipeline*, podemos usar o índice para processar a primeira operação através de uma seleção eficaz do inventário, i.e., (*store_no* = 10). Armazenamos o resultado em uma relação temporária, e aplicamos este a segunda operação de seleção, i.e., *item_code* = 1001.

Usando a técnica *pipeline*, dispensamos o uso de uma relação temporária. Ao invés disso, aplicamos a segunda operação de seleção (*item_code* = 1001) para cada fila devolvida pela primeira operação de seleção (*store_no* = 10).

Geralmente, um *pipeline* é implementado como um processo separado ou um serviço dentro do sistema de administração de banco de dados. Cada *pipeline* leva um *stream* de linhas com sua contribuição e cria um *stream* de linhas como saída. Um *buffer* é usado para cada par de operações adjacentes segurar o início das linhas passadas da primeira para a segunda operação. Uma desvantagem em usar *pipelining* é que isso abre operações que não necessariamente estão disponíveis para serem processadas e pode restringir a escolha de algoritmos que serão utilizados.

¹ Arquitetura do processador central que permite a execução de inúmeras atividades ao mesmo tempo

7. Consulta Otimizada do JavaDB

Nesta seção faremos uma avaliação do *optimizer* do **JavaDB** e os discutiremos os assuntos de desempenho na execução das declarações **DML**. JavaDB é usado freqüentemente como uma escolha para executar mais rapidamente a recuperação de registros no banco de dados. Por exemplo, usar um índice para um rápido *lookup* em entradas específicas, ou percorrer uma tabela inteira para obter as linhas necessárias. Para declarações que requereriam um *join*, é possível escolher qual tabela para examinar primeiro (ordem do *join*) e como unir as tabelas (estratégia do *join*). **Otimizar** significa que JavaDB faz a melhor escolha para recuperar as linhas, ordem do *join*, e uma melhor estratégia de *join*. A verdadeira otimização de consultas significa uma boa escolha embora como a consulta é escrita. O *optimizer* necessariamente não faz a melhor escolha.

7.1. Índices do Java DB e Caminhos de Acesso

Se um índice estiver definido para uma coluna ou colunas, o *optimizer* de consulta pode achar os dados nas colunas mais rapidamente. **JavaDB** automaticamente cria índices para as chaves primárias, chaves estrangeiras, e *constraints* únicas. Estes índices sempre estarão disponíveis ao *optimizer*, como também os que serão criados explicitamente com o comando CREATE INDEX. O modo como **JavaDB** obtém aos dados, por um índice ou diretamente pela tabela é chamado de **caminho de acesso**. Considere nosso banco de dados da Loja Orgânica. Para os exemplos nesta seção, consideraremos as seguintes tabelas:

Tabela *store* - Chave primária com o campo *number*.

Tabela *item* - Chave primária com o campo *code*.

Tabela *inventory* - Chave composta com os campos *store_no* e *item_code*

Um índice guarda todo valor da coluna, e dados para recuperar as linhas. Quando um índice inclui mais de uma coluna, como no caso da tabela *inventory*, a primeira coluna é a principal pelo qual as entradas são ordenadas. Se houver mais de um valor da primeira coluna, essas entradas são ordenadas para a segunda coluna. Por exemplo, quando são executadas as consultas mostradas abaixo, o índice definido para a chave primária da tabela *store* é usado.

```
SELECT * FROM store WHERE number = 20;
SELECT * FROM store WHERE number < 30;
SELECT * FROM store WHERE number >= 30;
```

Se **JavaDB** usa um índice para uma declaração, isto é dito que a declaração é otimizável. Declarações que possuem cláusulas WHERE que indicam começo e condições de parada permitem que **JavaDB** utilize o índice. Quer dizer, contam para o **JavaDB** o ponto ao qual iniciar e terminar sua procura do índice. Um índice percorre do começo ou são chamadas condições de parada de um índice indexando a procura.

Alguns predicados são diretamente otimizáveis se proverem um começo claro e pontos de término de parada e se:

- Usar uma referência de uma simples coluna para uma coluna (a coluna não deveria ser usada em uma expressão ou chamada de método).
- Isto refere-se a uma coluna que é a primeira ou só uma coluna no índice.
- A coluna é comparada a uma constante ou para uma expressão que não inclui colunas na mesma mesa. Os operadores de comparação que devem ser usados são:
 - =, <=, >=, <, >
 - IS NULL
- Se os predicados não forem diretamente otimizáveis, é dito que eles são indiretamente predicados de otimização, e são transformados interiormente em predicados de otimização. É dito que os seguintes operadores de comparação são indiretamente predicados de otimização:
 - BETWEEN

- LIKE (em certas situações)
- IN (em certas situações)

Alguns exemplo de consultas que não são otimizáveis são mostradas no código a seguir.

```
SELECT * FROM store WHERE number <> 30
-- por causa do operador não otimizável <>

SELECT * FROM INVENTORY WHERE item_code = 1004
-- até mesmo se item_code está definido no índice que não é o primeiro
```

Um índice cobre os meios da consulta de todas as colunas que especificados na consulta e fazem parte do índice. Estes são todos referenciados das colunas pela consulta; não só as colunas na cláusula WHERE. Um índice ao fazer isto acelera a execução da consulta mesmo se não há começo definido ou pontos de parada para um índice procurar.

Se uma cláusula WHERE contém pelo menos um predicado de otimização, então é otimizável. É procurado um índice para percorrer e poder usar qualificadores que mais adiante restringem o conjunto do resultado. Considere a consulta do código abaixo. O segundo predicado não é otimizável mas o primeiro é. O segundo predicado é usado por **JavaDB** para avaliar as entradas no índice e como percorrê-lo.

```
SELECT * FROM store WHERE number < 50 AND number <> 30
```

Os seguintes operadores de comparação são qualificadores válidos:

```
=, <, <=, >, >=, <>
IS NULL, IS NOT NULL
BETWEEN
LIKE
```

Para união é especificada com a palavra chave JOIN é otimizável o que significa que **JavaDB** pode usar um índice na tabela interna da união porque o começo e condições de parada estão sendo providas implicitamente pelas filas da tabela exterior. Também, a união construída usando os predicados tradicionais também são otimizáveis.

Se a declaração devolver virtualmente todos os dados da tabela, é melhor percorrer uma tabela em lugar de usar o índice. Neste caso, **JavaDB** pode evitar o passo intermediário de recuperar as linhas dos valores de *lookup* de índice. Considere a consulta mostrada abaixo:

```
SELECT * FROM item WHERE code < 2000
```

Para a tabela *item*, a maioria da coluna *item_code* é menos que 2000. Porém, para a questão mostrada no código abaixo, **JavaDB** usa um índice:

```
SELECT * FROM item WHERE code < 1010
```

JavaDB precisa manter os índices. Isto significa que quando são inseridas ou apagadas linhas, o sistema deve inserir ou apagar as linhas em todos os índices da tabela. Se uma atualização em uma coluna tem um índice para execução, então uma atualização nos índices também precisa ser feita. Isto significa que ao ter muitos índices pode acelerar as declarações SELECT mas em compensação piorar os tempos das declarações de inserção, atualização e exclusão.

7.2. Joins e Performance

Como uma revisão, um *join* recupera os dados de duas ou mais tabelas que usam uma ou mais colunas fundamentais para cada tabela. O desempenho de uma operação *join* varia amplamente. Fatores que afetam o desempenho uma operação *join* é a ordem, a estratégia e os índices.

Ordem do Join

JavaDB percorre as tabelas em uma ordem particular, i.e., acessa primeiro primeiro linhas em uma tabela, e esta é chamada de tabela exterior. Então, para cada linha qualificativa na tabela exterior, emparelha a coluna na segunda tabela que é conhecida como a tabela interna. **JavaDB**

só acessa uma vez a tabela exterior. Porém, a tabela interna pode ser acessada muitas vezes. As regras para uma ordem são as seguintes:

- Se uma operação *join* não tem nenhuma restrição na cláusula WHERE que limitaria o número de linhas devolvidas em um das tabelas, as seguintes regras se aplicam:
 - Se só uma tabela estiver com um índice unida na coluna ou colunas, é melhor fazer aquela tabela como uma tabela interna porque para cada um dos muitos *lookups* de tabela interno, **JavaDB** poderá usar um índice em vez de percorrer a tabela inteira.
 - Desde que os índices em tabelas internas são acessados muitas vezes, se o índice em uma tabela for menor que o índice em outra, a tabela com o menor índice deveria ser a tabela interna. Porque índices menores (ou tabelas) podem ser colocados em cache (detendo a memória, enquanto permite evitar caros I/O para cada repetição).
- Por outro lado, se a questão tiver restrições na cláusula WHERE para uma tabela que devolveria algumas linhas desta, é melhor a tabela restringida ser a tabela exterior. Deste modo, o sistema de qualquer maneira terá que ir para a tabela interna somente algumas vezes.

Estratégias do Join

Existem duas estratégias de *Join* utilizadas no JavaDB:

1. **Nested Loop.** Isto é o tipo mais comum de estratégia *join* em JavaDB. Para cada linha qualificativa na tabela exterior, usa um caminho de acesso apropriado para achar as linhas emparelhando na tabela interna.
2. **Hash Join.** **JavaDB** constrói uma tabela de *hash* que representa todas as colunas selecionadas na tabela interna. Para cada linha qualificativa na tabela exterior, executa um *lookup* rápido nesta tabela de *hash* para obter os dados internos. Neste caso, percorrer uma única vez a tabela interna ou índice, e isso é quando a tabela de *hash* é construída. Este tipo de estratégia *join* é preferível em situações nas quais a tabela interna contém valores que não são iguais, e há muitas linhas qualificativas na tabela exterior. Requer que a cláusula WHERE é o *equijoin* de otimização, i.e.:
 - Deve usar o operador de igualdade (=) para comparar as colunas na tabela exterior para as colunas na tabela interna.
 - Deve ter a referência das colunas que são referências de coluna simples como previamente discutido em predicados diretos de otimização.

A tabela de *hash* para um *hash join* é preso em memória. Se ficar grande, usará o disco como um *overflow*. A otimização faz uma forte estimativa na memória requerida para a tabela de *hash*. Se a estimativa for maior que o limite de memória, optará por usar uma estratégia de *nested loop*.

Otimização Baseada em Custo

A otimização de consulta de **JavaDB** toma decisões baseadas em custo para determinar:

- **Qual índice (se qualquer) usar em cada tabela em uma consulta que define a escolha do optimizer de caminho de acesso.** Isto pode depender do número de linhas que serão lidas. Isto irá escolher um caminho que requer poucas colunas para ler. Se tem uma operação *join*, também dependerá de uma ordem. Às vezes, é possível conhecer exatamente quantas linhas serão lidas. Em outros tempos, faz uma suposição educada que faz uso de seletividade e estatísticas da cardinalidade que serão discutidas mais tarde.
- **Qual ordem usar.** A ordem pode afetar qual índice será usado. O *optimizer* pode escolher um índice como o caminho de acesso para uma tabela se for a tabela interna; não se é a tabela exterior, e que não há nenhuma qualificação adicional. Escolhe só uma ordem de tabelas dentro de uma simples cláusula FROM. A maioria usa a palavra chave JOIN é aplainado em uma simples união, de forma que escolhe a ordem. Não escolhe a ordem para a união exterior; usa uma ordem especificada na declaração. Ao selecionar uma ordem, leva em conta:
 - Tamanho de cada tabela

- Índices disponíveis em cada tabela
- Se um índice em uma tabela é útil em uma ordem particular
- Número de linhas e páginas procuradas para cada tabela em uma ordem
- **Qual estratégia usar.** O *optimizer* compara o custo de escolher um *hash* (se um *hash* for possível) para o custo de escolher um *nested loop* e escolhe a estratégia mais barata. A tabela de *hash* usará a memória e os meios que precisar para aproximar o tamanho das tabelas de *hash*. Possui um limite superior no tamanho de uma tabela na qual considerará um *hash*. Não considerará um *hash* para uma declaração se calcular que o tamanho da tabela de *hash* excederia o limite do uso de memória para uma tabela. Escolherá, então, uma estratégia *nested loop*.
- **Evitar a escolha adicional.** Algumas declarações de SQL exigiriam ordenar dados que são incluídos em ORDER BY, GROUP BY e DISTINCT. Funções de agregação como MIN() e MAX() também necessita ordenar os dados. **JavaDB** pode às vezes evitar a escolha usando os seguintes passos:
 - **Custo baseado em ORDER-BY.** Se uma consulta em uma única tabela tem uma cláusula de ORDER-BY em uma única coluna, e há um índice naquela coluna, então ordenações podem ser evitadas desde que **JavaDB** possa usar o índice como um caminho de acesso.
 - **Evitando o Não-custo Baseado em Ordenação ou Filtro de Tuplas.** Declarações que usam as palavras chaves DISTINCT ou GROUP-BY, precisa executar dois passos separados:
 - O primeiro passo envolve a escolha das colunas selecionadas. Então, pode descartar as filas duplicadas ou as filas agrupadas. JavaDB pode evitar a escolha com filtro, i.e., as filas são lidas em uma ordem útil.
 - Para a palavra chave DISTINCT, JavaDB pode filtrar simplesmente valores fora de duplicatas quando são achados os resultados e voltados cedo para o usuário. O filtro é aplicado quando as seguintes condições forem estabelecidas:
 - A cláusula SELECT é composta inteiramente de colunas com simples referências e constantes
 - Simples referências de coluna vêm da mesma tabela, e o *optimizer* pode escolher a tabela para ser a tabela externa no bloco da consulta.
 - Um índice é escolhido como o caminho de acesso para a tabela.
 - As referências de simples coluna na cláusula SELECT, e qualquer referência de coluna simples da tabela tem predicados de igualdade, é um prefixo do índice que o *optimizer* pode selecionar como o caminho de acesso para a tabela.

Veja o código a seguir para o **JavaDB** versão 10.3:

```
CREATE TABLE t1(c1 INT, c2 INT, c3 INT, c4 INT)
CREATE INDEX i1 ON t1(c1)
CREATE INDEX i1_2_3_4 ON t1(c1, c2, c3, c4)

--Este é o caso mais comum no qual são aplicados filtros:
SELECT DISTINCT c1 FROM t1

--Predicados de igualdade permitem os seguintes filtros:
SELECT DISTINCT c2 FROM t1 WHERE c1 = 5
SELECT DISTINCT c2, c4 FROM t1 WHERE c1 = 5 and c3 = 7
--As colunas não precisam estar na mesma ordem como o índice
SELECT DISTINCT c2, c1 FROM t1
```

Para a cláusula GROUP BY, JavaDB pode agregar um grupo de linhas até que um novo conjunto de linhas seja descoberto e retorna um resultado mais rápido para o usuário. Filtros são aplicados quando as seguintes condições foram conhecidas:

- Todas as colunas do agrupamento vêm da mesma tabela e o *optimizer* escolher a tabela da consulta ser a tabela externa no bloco desta.
- O *optimizer* escolheu um índice como o caminho de acesso para a mesa em questão.
- As colunas de agrupamento, mais que qualquer referência de coluna simples da tabela que tem predicados de igualdade neles, são um prefixo do índice que o *optimizer* seleciona como o caminho de acesso para a tabela.

Veja o código a seguir para o **JavaDB** versão 10.3:

- **Otimização com MIN() e MAX().** O *optimizer* sabe que pode evitar interação por todas as linhas fontes em um resultado para computar um MIN() ou MAX() agregado quando dados já estão na ordem certa. Quando são garantidos dados estar na ordem certa, **JavaDB** pode imediatamente ir para a menor (mínima) ou maior (máxima) linha. As condições seguintes devem ser satisfeitas:
 - O MIN() ou MAX() é a única entrada na lista SELECT.
 - O MIN() ou MAX() está em uma referência de coluna simples e não em uma expressão.
 - Para o MAX(), não deve haver uma cláusula WHERE.
 - Para MIN():
 - A tabela de referência e a tabela externa no *optimizer* escolheu uma ordem para o bloco de consulta.
 - O *optimizer* escolheu um índice que contém a coluna de referência como o caminho de acesso.
 - A coluna de referência é a primeira coluna fundamental naquele índice OU a coluna de referência é uma coluna fundamental naquele índice e os predicados de igualdade existem em todas as colunas de chave antes da referência da coluna simples naquele índice.

Exemplos serão mostrados abaixo.

```
-- Por exemplo, o optimizer pode usar esta otimização para as seguintes
-- questões (se o optimizer usa o apropriado índices como os caminhos de
-- acesso):
```

```
-- índice com o número
SELECT MIN(number) FROM store
```

```
-- índice com o número
SELECT MAX(number) FROM store
```

```
-- índice com o segment_number, flight_id
SELECT MIN(item_code) FROM inventory
WHERE store_no = 10
```

- Em uma seleção de *bulk fetch*. Um *bulk fetch* vai buscar mais que um coluna de cada vez. É mais rápido que recobrar uma linha de cada vez quando um número grande de linhas for qualificada para cada percorrer a tabela ou índice. Utiliza uma memória extra para segurar as linhas pre-buscar. É evitado em situações nos quais a memória é escassa. É automaticamente trocado os cursores de *updatable*, dos *hash join*, para as declarações que devolvem uma única linha, e para subqueries. **JavaDB** vai buscar 16 linhas de cada vez por padrão.

7.3. Seletividade e Estatísticas de Cardinalidade

Na ordem para o *optimizer* decidir em um caminho de acesso para uma tabela particular, i.e., se usar o índice ou percorrer a tabela, deve determinar o número de linhas que serão percorridas nos discos. Saber exatamente o número de linhas que serão percorridas, ou calcular este número.

Como o *optimizer* determina o número de linhas da tabela?

O catálogo do sistema do JavaDB conta as linhas de uma tabela. O sistema mantém isto automaticamente. E é usado pelo *optimizer* determinar o número de filas que será percorrido nos discos. Quando um sistema não fizer paralisação de empresas corretamente, a conta das linhas armazenadas ficarão inexatas. Quando um sistema executar uma paralisação de empresas por ordem, irá atualizar a conta de linha armazenada. As contas das linhas são automaticamente atualizadas quando o **JavaDB** executar uma consulta que percorre toda a base de dados. Considere a declaração SELECT a seguir:

```
SELECT * FROM store
```

Executando os conjuntos de consulta seguintes a conta de tabela para guardar o valor correto. JavaDB também fixa a conta de linha armazenada em uma tabela para um valor correto sempre que um usuário criar um novo índice ou chave primária, únicas, ou chave estrangeira na tabela.

```
SELECT * FROM inventory
WHERE store_no = 10;
```

Como o *optimizer* estima o número de linhas da tabela?

Quando um índice estiver disponível, o *optimizer* calcula o número de linhas que serão percorridas pelo disco. A precisão da estimativa está baseado no tipo consulta que inicia a otimização, especificamente, examina a condição de consulta conhecida ou examina com condição de procura desconhecida.

1. **Consultas com condição de procura conhecida.** É dito que uma questão sabe a condição de procura quando o começo e condições de parada são conhecidas a compile tempo. Neste caso, o *optimizer* usa o próprio índice para calcular o número de filas que serão esquadrinhadas de disco. Considere a consulta abaixo:

```
SELECT * FROM inventory
WHERE store_no = 10;
```

A condição de pesquisa tem um valor de procura que é conhecido (store_no = 10). O *optimizer* pode fazer uma estimativa precisa baseado no índice definido na chave primária (store_no, item_code) para a tabela *inventory*. O Campo *store_no* é a primeira coluna da chave composta. Também, se o índice é sem igual, e a cláusula WHERE envolve um operador de igualdade (=) ou comparação IS NULL, a otimização conhecida só uma única linha será percorrida no disco. Considere a consulta abaixo. Neste exemplo, está a chave primária da tabela *store*. Então, o sistema indexado na chave primária é considerado um índice sem igual. O *optimizer* sabe que esta questão devolverá uma única linha.

```
SELECT * FROM store
WHERE number = 10
```

2. **Examinar condição de consulta desconhecida.** Há questões que têm condições de procura desconhecidas como a cláusula WHERE que pode conter parâmetros dinâmicos que só são conhecido durante tempo de execução, ou quando a declaração envolve um *join*. O *optimizer* fará uma suposição no número de linhas devolvidas usando os dados de **seletividade** das tabelas para uma cláusula WHERE em particular. Seletividade recorre à fração de linhas que serão devolvidas da tabela para a cláusula WHERE em particular. Se a seletividade para uma operação de procura particular é 0.10, e a tabela contém 100 linhas, as estimativas de *optimizer* devolve 10 linhas. Note que esta apenas uma suposição.

JavaDB determina a selectividade para a cláusula WHERE usando:

- **Seletividade de Estatísticas de Cardinalidade.** Estatísticas de Cardinalidade são computadas pelo JavaDB, e armazenadas nas tabelas do sistema. JavaDB usa isto se:
 - Existem as estatísticas
 - As colunas pertinentes na cláusula WHERE são colunas principais em um índice

- As colunas que usam o operador de igualdade são comparadas (=)
- As Estatísticas de Cardinalidade serão discutidas na próxima seção.
- **Seletividade de Suposições Hard-Wired.** Estas são suposições hard-wired para a JavaDB. Usa um número fixo que tenta descrever a porcentagem de filas que provavelmente serão devolvidas. Necessariamente poderia não corresponder à seletividade atual da operação em todo caso. A tabela mostra 5 a seletividade de várias operações para índice de procura quando os valores forem desconhecidos com antecedência e estatísticas não são usadas.

<i>Operator</i>	<i>Selectivity</i>
operadores relacionais como =, >=, <=, <, <> quando o tipo de dado do parâmetro é um lógico	0.5 (50%)
Outros operadores (exceto para IS NULL e IS NOT NULL) quando o tipo de dado do parâmetro é um lógico	0.5 (50%)
IS NULL	0.1 (10%)
IS NOT NULL	0.9 (90%)
Operador =	0.1 (10%)
>, >=, <, <=	0.33 (3%)
<> comparado com um tipo não-lógico	0.90 (90%)
LIKE transformado para um predicato LIKE	1.0 (100%)
Operadores >= e < quando transformado internamente com um LIKE	0.25 (0.5 x 0.5)
Operadores >= e <= quando transformado internamente com um BETWEEN	0.25 (0.5 x 0.5)

Tabela 5: Suposições de Selectividade Hard-wired

O que são estatísticas de cardinalidade?

JavaDB calcula e armazena nas tabelas de sistema o seguinte:

- número de linhas em cada tabela
- número de valores únicos para um conjunto de colunas para as principais em uma chave de índice que também é conhecido como o cardinality

Colunas principais recorrem à primeira coluna, ou a primeira e segunda coluna, ou a primeira, segunda, e terceira coluna de um índice (e assim por diante). JavaDB não pode computar o número de colunas para as quais uma combinação das colunas não-principais é única.

São unidas as Estatísticas de Cardinalidade nas chaves de um índice quando este for criado.

Java cria automaticamente estatísticas novas ou atualizações existentes para as seguintes operações:

- Quando um índice novo em uma tabela não-vazia existente é criado. São automaticamente criadas estatísticas para o índice novo.
- Quando uma chave primária, constrangimento fundamental sem igual, ou estrangeiro para uma mesa non-vazia existente é somado. Se houver nenhum índice existente que pode ser usado para a chave nova ou constrangimento, JavaDB cria estatísticas automaticamente para os novos índices.
- Quando executar o procedimento de sistema SYSCS_UTIL.SYSCS_COMPRESS_TABLE. São criadas estatísticas automaticamente para todos os índices se estas já não existirem.
- Quando uma coluna que faz parte do índice de uma tabela é encerrada, são encerradas as estatísticas para o índice afetado. Estatísticas são automaticamente atualizadas para os outros índices na tabela.

É possível que estatísticas possam ser passadas porque existem poucos casos limitados que são automaticamente atualizados. As declarações de inserção, atualização, e exclusão não causam atualização das estatísticas. Deve estar atento que as estatísticas passadas podem reduzir a velocidade seu sistema, porque pioram a precisão do *optimizer* quanto as estimativas de seletividade.

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.