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

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

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

Configuração Mínima de Hardware

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

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

Configuração Recomendada de Hardware

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

Requerimentos de Software

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

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

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

Java™ DB System Requirements

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

Colaboradores que auxiliaram no processo de tradução e revisão

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

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

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

Coordenação do DFJUG

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

Agradecimento Especial

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

1. Objetivos

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 procidimental** onde o usuário especifica que dado quer antes de informar como ele será recuperado

Nesta seção discutiremos o calculo 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 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. Algebra Relacional

Algebra 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 fas com que a saída de uma operação seja a entrada de outra operação. Esta propriedade é conhecida como fechamento (clousure), isto é, expressões podem ser aninhadas na algebra relacional similarmente às operações aritiméticas.
Algebra 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 relaçao simples R e define a relação que contem 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 contem unicamente a linha `PhoneCard` ou registros com valor igual a 500.

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

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

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

Exemplo 2: Produz a lista de todos os cartões de telefone "`PhoneCard`" mostrando apenas o numero 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 combiações de cardNo, suppleirId, value, recipientID e dataLoaded de todas as contas.

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

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.

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

- ***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 compátivel 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.ai θ S.bi, onde θ pode ser uma operação de comparação (<, >, =, <=, >=, ~=)

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

Equi-join(junção de igualdades)

No caso do predicado de F conter apenas igualdades (=), teta-jooin 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.

$$R \square S$$

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**.

$$R \square S$$

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.

$$R \rhd_F S$$

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-escrevida usando as operações de projeção e de junção.

$$R \rhd_F S = \Pi_A(R \square_F S)$$

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.

$$R \square S$$

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

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

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 preposição 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 | P(x)}

Os predicados podem ser conectados usando conectivos como:

- ∧ (AND)
- ∨ (OR)
- ~ (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 | 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 | E.dataDeContrato > 'January 1, 2005'}

Dois qualificadores são usados com nessa fórmula.

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

GAMA DE E É EMPREGADO

∃ E {E.Numero = 5001}

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

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

GAMA DE E É EMPREGADO

∀ E {E.DataContrato ~= '15 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, ..., t_n são constantes ou variáveis então a formula será P(t_1,t_2,...,t_n).
2. Se t_1 e t_2 são constantes ou qualquer uma variavel do mesmo dominio e Θ um dos operadores (<, <=, >, >=, =, ~=) 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 $\forall X(F)$ tambem são formulas.

- **Dominio orientado.** Variaveis que são usados para pegar valores de dominio ao inves de relações de tuplas. Se $P(d_1, d_2, ..., d_n)$ são pos no predicado com variaveis $d_1, d_2, ..., d_n$, então

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

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 relaticional orientado a dominio, determinando se valores pertencem a uma relação. A expreção *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.

{UltimoNome, PrimeiroNome, DataContrato | ∃ DataContrato (Empregado(UltimoNome, PrimeiroNome, DataContrato) ∧ DataContrato > '1 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 *(Empregado(UltimoNome, PrimeiroNome, 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 dados[1]
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-simbolos

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

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

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 eas 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 inciar 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.

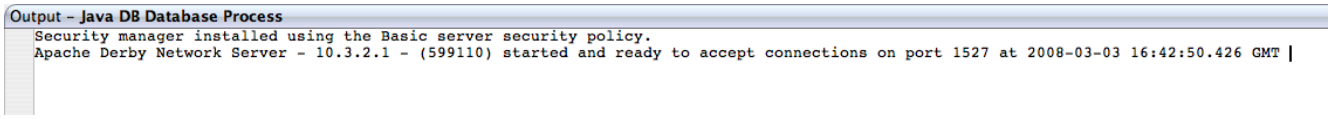

Figura 1: Mensagem inicial do JavaDB

4.2. Criando o banco de dados

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

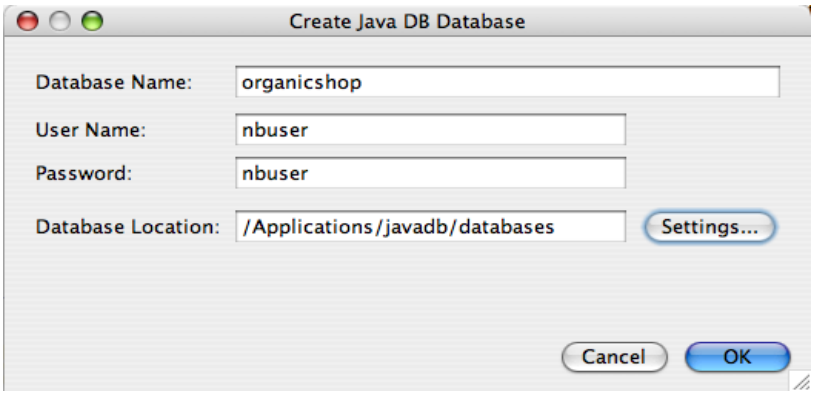

Figure 2: Janela da criação do banco dados Java DB

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.

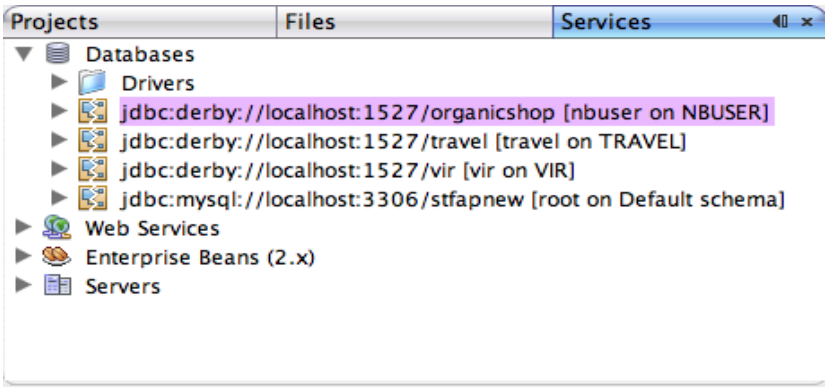

Figure 3: O banco de dados Organic Shop criado

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/organicshop[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.

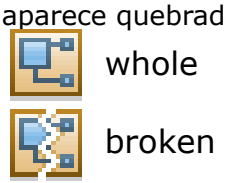

Figura 4: Indicador de Conexão com o banco de dados

4.4. Desconectando o NetBeans do Banco de dados

Clique com o botão direito do mouse na opção:
`(jdbc:derby://localhost:1527/organicshop[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/organicship[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/organicshop[nbuser on NBUSER]) Selecione a opção Execute. A Figura 6 mostra o SQL Editor.

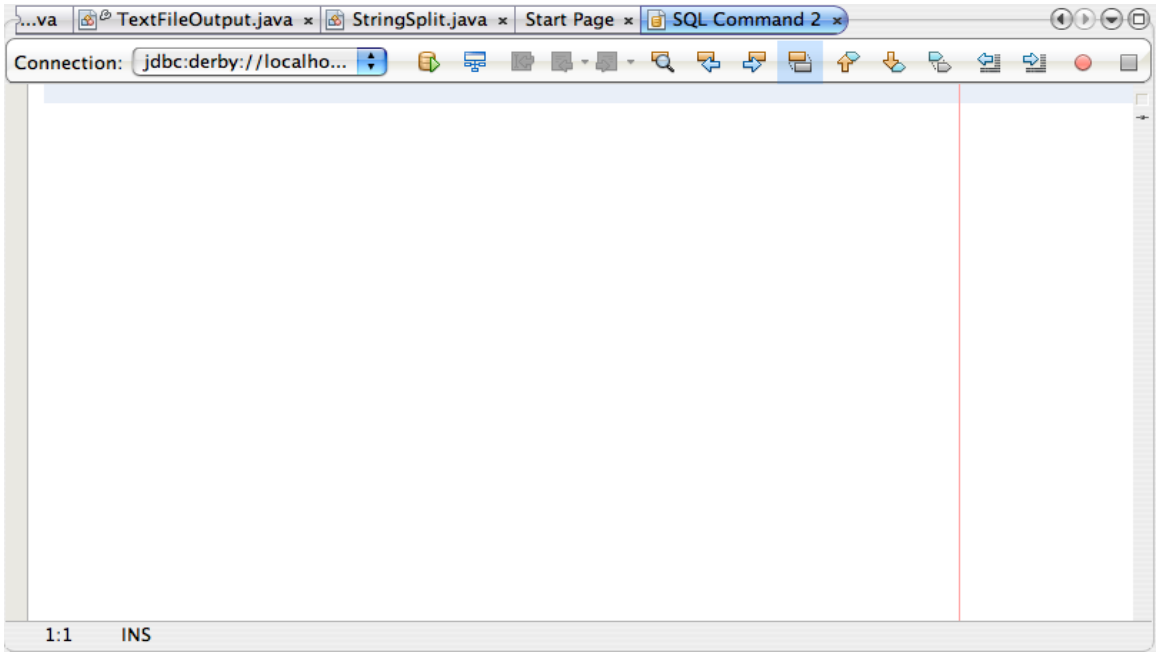

Figura 6: SQL Editor no NetBeans

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.

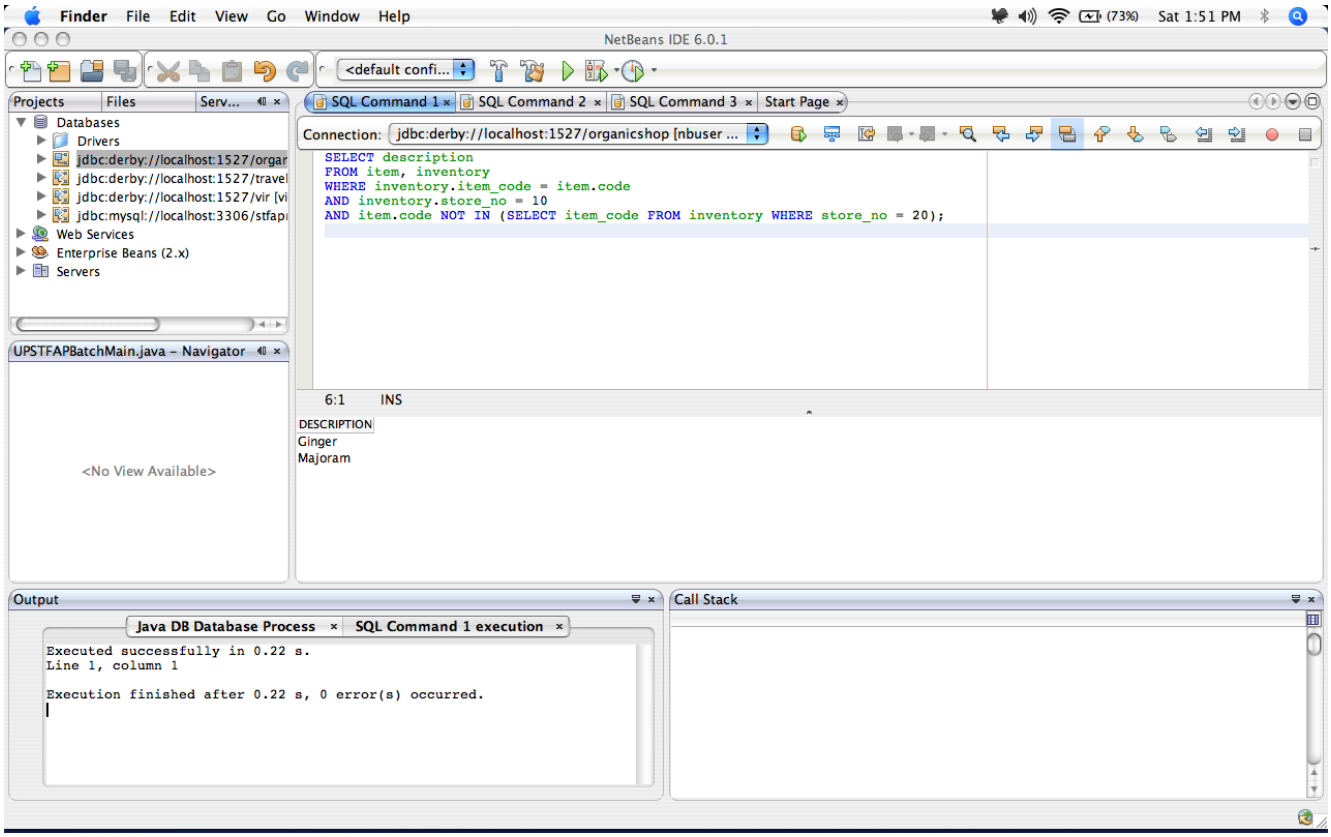

Figura 7: Executando Comandos SQL Statement no NetBeans

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 (foriegn 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 CHEK. 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,
  addreess 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 registras 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 freqüentemente utilizados em instruções SQL. Consistem em:

- instrução SELECT
- instrução INSERT
- instrução DELETE
- instrução UPDATE

6.1. Instrução SELECT

A instrução SELEC 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 freqüentemente 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:
	● % (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 '='`
		`expressãoCaracteres [NOT] LIKE expressãoCaracteresComCoringa [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

	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:
	`WHERE date_hired NOT IN (SELECT`
	`date_hired FROM employee`
	`WHERE Store = 10`
	`)`
		`expressão [NOTE] IN`
		`subConsultaDaTabela |`
		`expressão [NOT] IN`
		`(expressão [, expressão]*)`
		`}`

EXISTS	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:
	`WHERE EXISTS (SELECT *`
	`FROM employees`
	`WHERE date_hired > DATE('2001-01-01'))`
		`[NOT] EXISTS`
		`SubConsultaDaTabela`

ALL
ANY
SOME
	Esses são conhecidos como comparação quantificada. Uma comparação quantificada é umas operações lógicas (<, =,>, <=>, =, <>) com ALL ou ANY ou SOME aplicados.
	Se ALL for usado, a comparação deve ser verdadeira para todos os valores devolvidos pela sub-consulta da tabela.
	Se ANY ou SOME forem usados, a comparação deve ser verdadeira para pelo menos um valor da sub-consulta da tabela. Example:
	`WHERE ave_qty < ALL (SELECT amount/500 FROM Inventory)`
		`expressão operadorDeComparação`
		`{`
		`ALL | ANY | SOME`
		`}`
		`subConsultaDaTabela`

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	É uma função de agregação que conta o número de linhas acessadas na expressão. É permitida em todos os tipos de expressões. Sintaxe: `COUNT([DISTINCT|ALL] expression)`
MIN	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: `MIN([DISTINCT|ALL] expression)`
MAX	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: `MIN([DISTINCT|ALL] expression)`

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.