



Computação

Banco de Dados

Cicero Tadeu Pereira Lima França
Joaquim Celestino Júnior



Geografia



História



Educação
Física



Química



Ciências
Biológicas



Artes
Plásticas



Computação



Física



Matemática



Pedagogia



Computação

Banco de Dados

Cicero Tadeu Pereira Lima França
Joaquim Celestino Júnior

2ª edição
Fortaleza - Ceará



2015



Geografia



História



Educação
Física



Química



Ciências
Biológicas



Artes
Plásticas



Computação



Física



Matemática



Pedagogia

Copyright © 2015. Todos os direitos reservados desta edição à UAB/UECE. Nenhuma parte deste material poderá ser reproduzida, transmitida e gravada, por qualquer meio eletrônico, por fotocópia e outros, sem a prévia autorização, por escrito, dos autores.

Editora Filiada à



Presidenta da República
Dilma Vana Rousseff
Ministro da Educação
Renato Janine Ribeiro
Presidente da CAPES
Carlos Afonso Nobre
Diretor de Educação a Distância da CAPES
Jean Marc Georges Mutzig
Governador do Estado do Ceará
Camilo Sobreira de Santana
Reitor da Universidade Estadual do Ceará
José Jackson Coelho Sampaio
Vice-Reitor
Hidelbrando dos Santos Soares
Pró-Reitor de Pós-Graduação
Jerffeson Teixeira de Souza
Coordenador da SATE e UAB/UECE
Francisco Fábio Castelo Branco
Coordenadora Adjunta UAB/UECE
Eloísa Maia Vidal
Direção do CED/UECE
José Albio Moreira de Sales
Coordenação da Licenciatura em Computação
Francisco Assis Amaral Bastos
Coordenação de Tutoria da Licenciatura em Computação
Maria Wilda Fernandes Felipe
Editor da EdUECE
Erasmio Miessa Ruiz
Coordenadora Editorial
Rocylânia Isídio de Oliveira
Projeto Gráfico e Capa
Roberto Santos
Diagramador
Francisco Oliveira
Revisão Ortográfica
Fernanda Ribeiro

Conselho Editorial

Antônio Luciano Pontes
Eduardo Diatahy Bezerra de Menezes
Emanuel Ângelo da Rocha Fragoso
Francisco Horácio da Silva Frota
Francisco Josênio Camelo Parente
Gisafran Nazareno Mota Jucá
José Ferreira Nunes
Liduina Farias Almeida da Costa
Lucili Grangeiro Cortez
Luiz Cruz Lima
Manfredo Ramos
Marcelo Gurgel Carlos da Silva
Marcony Silva Cunha
Maria do Socorro Ferreira Osterne
Maria Salete Bessa Jorge
Sílvia Maria Nóbrega-Therrien

Conselho Consultivo

Antônio Torres Montenegro (UFPE)
Eliane P. Zamith Brito (FGV)
Homero Santiago (USP)
Ieda Maria Alves (USP)
Manuel Domingos Neto (UFF)
Maria do Socorro Silva Aragão (UFC)
Maria Lírida Callou de Araújo e Mendonça (UNIFOR)
Pierre Salama (Universidade de Paris VIII)
Romeu Gomes (FIOCRUZ)
Túlio Batista Franco (UFF)

Dados Internacionais de Catalogação na Publicação
Sistema de Bibliotecas
Biblioteca Central Prof. Antônio Martins Filho
Meirilane Santos de Moraes Bastos – CRB-3 / 785
Bibliotecária

F814b França, Cícero Tadeu Pereira Lima.
Banco de dados / Cícero Tadeu Pereira Lima França, Joaquim Celestino Júnior. – 2. ed. – Fortaleza, CE : EdUECE, 2015.
117 p. : il. ; 20,0cm x 25,5cm. (Computação)
Inclui referências.
ISBN:
1. Banco de dados. I. Celestino Júnior, Joaquim. II. Título.
CDD : 001.64

Editora da Universidade Estadual do Ceará – EdUECE
Av. Dr. Silas Munguba, 1700 – Campus do Itaperi – Reitoria – Fortaleza – Ceará
CEP: 60714-903 – Fone: (85) 3101-9893
Internet: www.uece.br – E-mail: eduece@uece.br
Secretaria de Apoio às Tecnologias Educacionais
Fone: (85) 3101-9962

Sumário

Apresentação	5
Capítulo 1 – Visão geral sobre Banco de Dados.....	7
1. Banco de Dados e Sistema Gerenciador de Banco de Dados	10
2. Modelos de Bancos de Dados	11
3. Arquiteturas de Banco de dados	13
3.1. Arquitetura Centralizada	13
3.2. Arquitetura Cliente-Servidor de Duas Camadas.....	14
3.3. Arquitetura Cliente-Servidor de Três Camadas	15
3.4. Arquitetura Distribuída.....	16
Capítulo 2 – Modelagem de Dados e Normalização	21
1. Sinopse do Projeto	23
2. Modelo Conceitual	24
2.1. Identificando os Tipos Entidades	24
2.2. Identificando os Tipos Relacionamento.....	25
2.3. Normalização	27
2.4. Modelo Lógico	36
Capítulo 3 – PostgreSQL e Modelagem Física	39
1. PostgreSQL.....	41
1.1. Instalando o PostgreSQL.....	41
2. Modelo Físico	48
2.1. pgAdmin.....	48
Capítulo 4 – Introdução a SQL.....	57
1. História	59
2. Grupos.....	61
2.1. Outros Grupos	61
3. SQL Editor.....	61
4. DDL – Parte 1	62
4.1. Comando CREATE TABLE.....	62
4.2. Comando ALTER TABLE	65
4.3. Comando DROP TABLE	67
4.5. DML.....	67
4.6. Comando SELECT – Parte 1.....	75
Capítulo 5 – SQL Avançada	87
1. Comando SELECT – Parte 2	89
1.1. Cláusula WHERE com Condições Complexas.....	89

1.2. Cláusula ORDER BY	91
1.3. Comando JOIN	93
1.4. Comando UNION	97
1.5. Funções Básicas	100
1.6. Cláusula GROUP BY	102
1.7. Cláusula DISTINCT	104
1.8. Operadores de Manipulação	105
1.9. Nested Queries	108
2. DDL – Parte 2	109
2.1. VIEW	109
2.2. STORE PROCEDURE	110
2.3. TRIGGER	113
2.4. DOMAIN	115
Sobre os autores.....	117

Apresentação

Desde os primórdios, a humanidade sempre procurou uma maneira para organizar seus conhecimentos de uma forma que a pesquisa por esses não demandasse grandes esforços. Tendo isso como base, podemos vê uma biblioteca como uma evolução desse conceito tão antigo quanto a própria consciência humana, uma vez que a ideia da biblioteca é justamente disponibilizar conhecimentos estruturados sistematicamente para facilitar a busca da informação desejada.

Com o advento do comércio, outros requisitos foram incluídos a essa necessidade, pois além de guardar o conhecimento ou informação das negociações, passou-se a ter a necessidade de poder incluir, modificar e excluir tais dados de uma maneira eficiente e eficaz. Um recurso usado por muito tempo para esses fins foi manter essas informações em arquivos (livros, fichas, etc.) e guardadas em local seguro. O armazenamento inadequado poderia causar transtornos e a perda de negócios.

O surgimento dos computadores e estudos relacionados ao armazenamento de dados levou as empresas a iniciarem uma migração gradativa para o mundo digital, inicialmente um recurso só disponível para grandes empresas devido os custos nos primeiros anos da computação moderna. Com a popularização dos computadores esses custos reduziram-se, ao ponto de toda empresa poder usufruir tais recurso.

As informações que migraram para o mundo computacional ficam armazenadas de maneira digital em banco dados. Esses bancos de dados foram multiplicados e nessa multiplicação apareceram diversos modelos rodando sobre diversas arquiteturas computacionais.

Nos dias atuais os bancos de dados se tornaram algum corriqueiro nos computadores, celulares, tablets, smartphone ou qualquer outro dispositivo computacional que tenham um aplicativo com necessite de um mínimo de armazenamento de dados.

Este livro apresentará o mundo dos bancos de dados de forma clara e simples, mostrando desde os conceitos básicos, indo do projeto à criação e uso de um banco de dados. No decorrer do livro serão apresentadas técnicas e tecnologias que possibilitarão a base necessário para o uso de banco de dados. Também será apresentado o PostgreSQL, uma SGBDR gratuito e de grande respeito na comunidade.

O livro está organizado em cinco capítulos. O primeiro capítulo disponibilizar um conhecimento geral sobre banco de dados, os modelos de banco de dados mais conhecidos e as arquiteturas mais populares. No capítulo 2 é iniciado um projeto de banco de dados e descrito técnicas para a criação do modelo conceitual, sua normalização e criação do modelo lógico. O capítulo 3 apresenta o PostgreSQL e mostra como criar um modelo físico no mesmo. Os capítulos 4 e 5 são destinados a SQL, mostrando desde um rápido histórico até seus comandos mais comuns segundo a ISO/IEC 9075.

O conteúdo apresentado neste livro destina-se principalmente a professores e alunos de graduação em Ciências da computação ou áreas afins, fornecendo uma base teórica e uma prática em projetos de banco de dados relacionais.

O autor

Capítulo

1

Visão geral sobre Banco de Dados

Objetivo

- Todas as informações encontradas no mundo computacional ficam armazenadas de maneira digital em banco de dados. Paralela a evolução da computação, os bancos de dados também evoluíram para atender as novas necessidades e exigências do mundo digital. Os modelos e arquiteturas de banco de dados mais comuns devem ser conhecidos, uma vez que eles apresentam várias opções para resolver as questões de armazenamento de dados conhecidas. O objetivo do capítulo é o conhecimento destes modelos.

Introdução

Até pouco tempo a maior parte das empresas mantinham as informações de cliente, fornecedores, produtos, entre outros, em arquivos de aço (figura 1.1) guardados em local seguro e com acesso físico controlado. Os dados do cliente, por exemplo, eram colocados em fichas de papel, e arquivados em ordem alfabética. Essas fichas continham informações de interesse da empresa, tais como nome, endereço, etc. Todos os clientes tinham uma ficha única que era armazenada de maneira apropriada em um arquivo físico. O armazenamento inadequado causava transtornos tanto para o cliente como para a empresa.

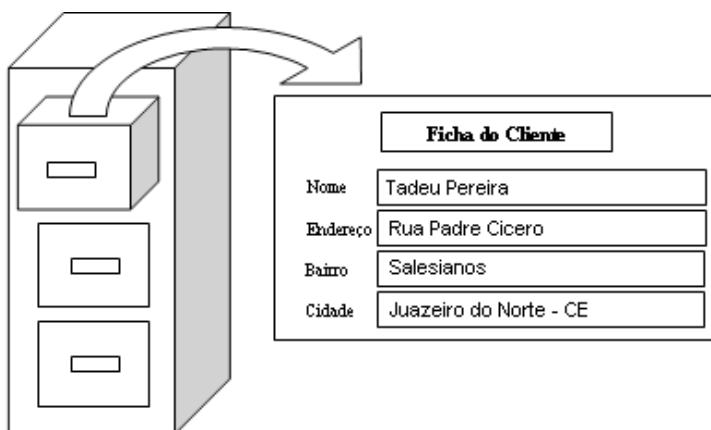


Figura 1.1 – Arquivo de aço antigo

Com advento dos computadores, as empresas migraram os dados para o mundo digital, facilitando e agilizando o seu manuseio. Muitos dos conceitos usados para manuseio físico de informações, conforme descrito acima, foram adaptados para os bancos de dados. A tabela 1.1 mostra um paralelo entre estes dois mundos.

Tabela 1.1

PARALELO ENTRE CONCEITOS	
Mundo Físico	Mundo Digital
Conjunto de Arquivos de aço	Banco de dados
Arquivo de aço	Tabela
Fichas de papel	Registro
Informações da ficha de papel	Campo

Um banco de dados pode ser visto como uma coleção de dados relacionados, que devem estar organizados para facilitar a busca e atualização desses dados. Os dados devem ter significado implícito e se referirem a fatos.

1. Banco de Dados e Sistema Gerenciador de Banco de Dados

Um banco de dados é um projeto delineado para o armazenamento de fatos que possuam um significado implícito, representando aspectos do mundo real. O conjunto de informações (dados) armazenado deve ter um significado efetivo que atenda um propósito específico.

Segundo [ELMASRI e NAVATHE, 2005] os dados devem ser providos de alguma fonte e ter “alguns níveis de interação com os eventos do mundo real e um público efetivamente interessado em seus conteúdos”.

Por outro lado, podemos definir ainda um Sistema Gerenciador de Banco de Dados (SGBD), que é um conjunto de programas que permite a criação e manipulação do banco de dados, facilitando os processos de definição, construção, manipulação e compartilhamento dos dados.

Para [RAMAKRISHNAN e GEHRKE, 2008] o uso de um SGBD proporciona várias vantagens. Uma delas é a possibilidade de “utilizar os recursos do SGBD para gerenciar os dados de uma forma robusta e eficiente”. Outra vantagem citada é o suporte indispensável do SGBD “à medida que cresce o volume de dados e o número de usuários”.

De uma maneira simplista, um banco de dados especifica os dados, as estruturas e as restrições, enquanto o SGBD gerencia a manipulação desses dados facilitando o acesso aos mesmos, extraindo essas responsabilidades dos usuários e aplicativos. A figura 1.2 mostra uma visão superficial do relacionamento entre o usuário, o SGBD e o banco de dados.

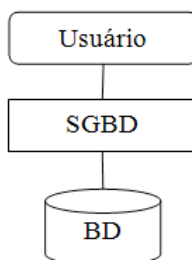


Figura 1.2 – Visão superficial de um SGBD e seus relacionamentos

2. Modelos de Bancos de Dados

À medida que aumentava o uso de banco de dados, foram surgindo necessidades que antes não eram vislumbradas, tais como:

- Compartilhamento dos dados armazenados;
- Controle de concorrência de uso sobre dados;
- Surgimento de novos paradigmas computacionais;
- Entre outros.

Com isso antigos modelos de banco de dados foram cedendo espaços para novos modelos, causando uma evolução nos mesmos. Na sequência são mostrados alguns modelos de banco de dados e suas características principais:

- **Banco de Dados Hierárquico:** Esse modelo foi muito utilizado nas primeiras aplicações de banco de dados; ele conecta registros numa estrutura de árvore de dados através de um relacionamento do tipo um-para-muitos (figura 1.3).

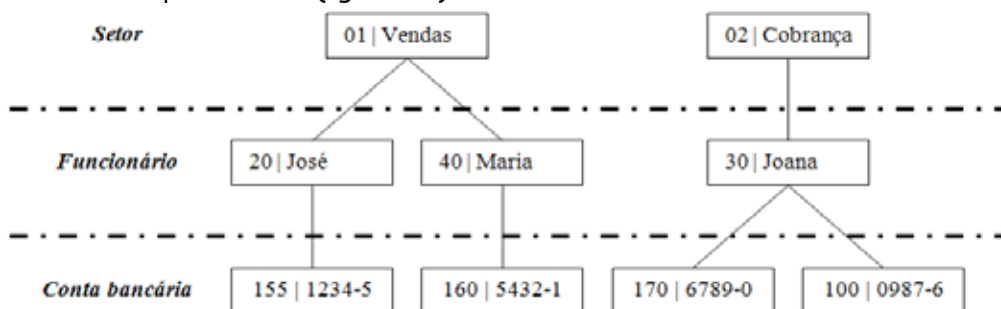


Figura 1.3 – Modelo de banco de dados hierárquico

- **Banco de Dados em Rede:** Semelhante ao anterior, mas nesse caso o relacionamento é do tipo muitos-para-muitos (figura 1.4).

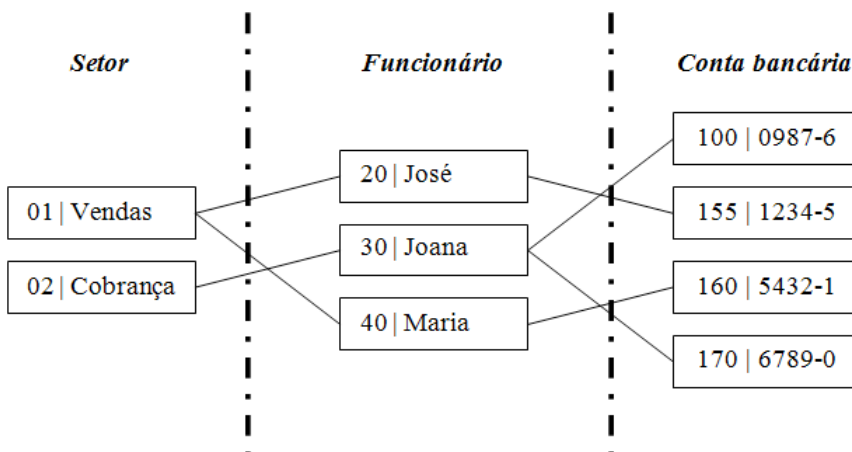


Figura 1.4 – Modelo de banco de dados em rede

- **Banco de Dados Relacional:** Concebido inicialmente para separar o modelo físico do conceitual, além de prover uma fundamentação matemática; sendo um modelo baseado na lógica e na teoria de conjuntos tornou-se o primeiro modelo de banco de dados formal; esse modelo é uma abstração que define o armazenamento, manipulação e recuperação dos dados estruturados na forma de tabelas; sendo largamente utilizado nos dias atuais (figura 1.5).

<i>Setor</i>		<i>Funcionário</i>			<i>Conta bancária</i>		
Código	Nome	Código	Nome	Setor	Código	Conta	Funcionário
01	Vendas	20	José	01	100	0987-6	30
02	Cobrança	30	Joana	02	155	1234-5	20
		40	Maria	01	160	5432-1	40
					170	6789-0	30

Figura 1.5 – Modelo de banco de dados relacional

- **Banco de Dados Orientado a Objetos:** Esse modelo foi criado pensando na necessidade do armazenamento e consulta de dados complexos, incorporando paradigmas já conhecidos da programação orientada a objetos (POO), tais como a abstração de dados, encapsulamento, herança e identificação de objetos (figura 1.6).

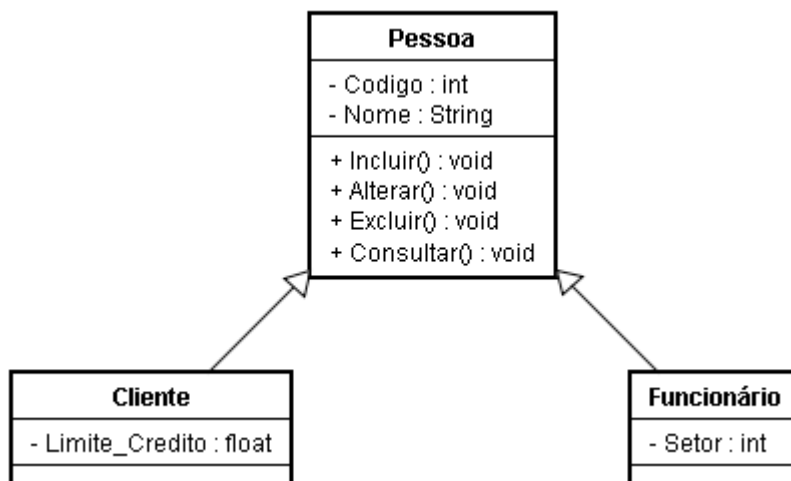


Figura 1.6 – Modelo de banco de dados orientado a objetos

- **Banco de Dados Objeto-Relacional:** Esse modelo é semelhante ao modelo relacional, mas alguns conceitos do modelo orientado a objeto foram incorporados; os esquemas do banco de dados dão suporte à criação e consulta de objetos, classes e herança (figura 1.7).

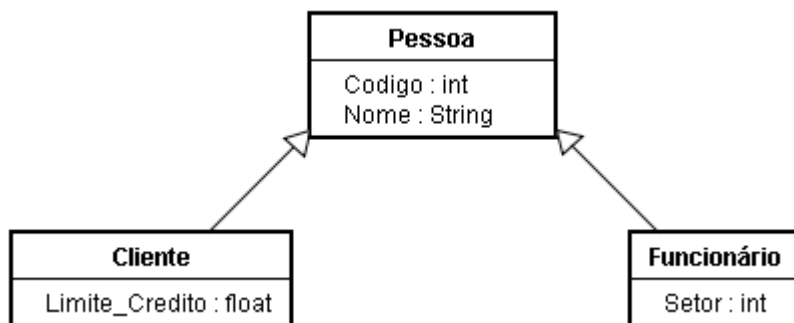


Figura 1.7 – Modelo de banco de dados objeto-relacional

3. Arquiteturas de Banco de dados

O dicionário Michaelis define arquitetura de uma maneira geral como:

1. Arte de projetar e construir prédios, edifícios ou outras estruturas; arquitetônica.
2. Constituição do edifício.
3. Contextura de um todo.
4. Intenção, projeto.

O mesmo dicionário define arquitetura em camadas como o “projeto de um sistema de computador em camadas, de acordo com a função ou prioridade”. Essa definição é a que melhor se aplica ao banco de dados.

Assim como os modelos de banco de dados citados anteriormente, no decorrer do tempo foram propostas varias arquiteturas. As principais arquiteturas propostas que foram e ainda são usadas são apresentadas nesta seção.

3.1. Arquitetura Centralizada

As arquiteturas dos bancos de dados centralizados utilizavam o conceito de concentrar nos mainframes o processamento das funções do sistema, programas de aplicação, programas de interface entre outras funcionalidades do SGBD.

Os usuários acessavam os sistemas via terminais de computadores que apenas exibiam resultados, mas não tinham poder computacional de processamento. Os processos em si eram executados remotamente no mainframe, que após processar o solicitado, enviava ao terminal as informações de exibição e os controles (figura 1.8).

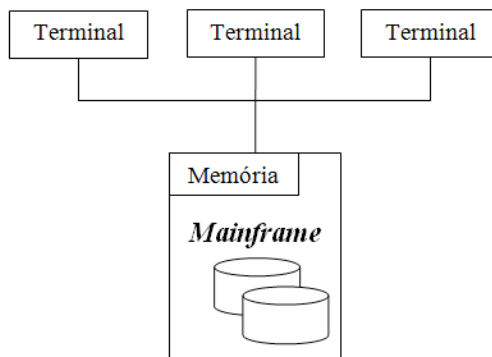


Figura 1.8 – Arquitetura centralizada

Com a queda nos preços do hardware, os usuários foram substituindo os antigos terminais por computadores pessoais (PCs) e workstations. No início dessa mudança os SGBDs ainda trabalhavam de forma centralizada, como se estivessem em terminais de computadores, mas aos poucos os SGBDs começaram a buscar o poder computacional disponíveis do lado do usuário. Essa mudança também causou mudanças na arquitetura centralizada, direcionando a mesma para uma arquitetura conhecida com Cliente-Servidor.

3.2. Arquitetura Cliente-Servidor de Duas Camadas

Essa arquitetura é uma evolução da arquitetura centralizada, pois graças à troca dos antigos terminais cliente por PCs e workstations o poder computacional do lado cliente aumentou, possibilitando passar parte do processamento para os mesmos e “desafogando” os servidores. Os programas de interface com o usuário e os de aplicação foram transpostos para o lado cliente.

Um cliente é geralmente uma máquina de usuário com poder computacional e com funcionalidade de interfaces, mas sempre que precisar acessar o banco de dados conecta-se com o servidor que viabiliza o acesso aos dados (figura 1.9). Geralmente têm-se máquinas com software cliente e outras com software servidor, mas podem-se ter máquinas com ambos.

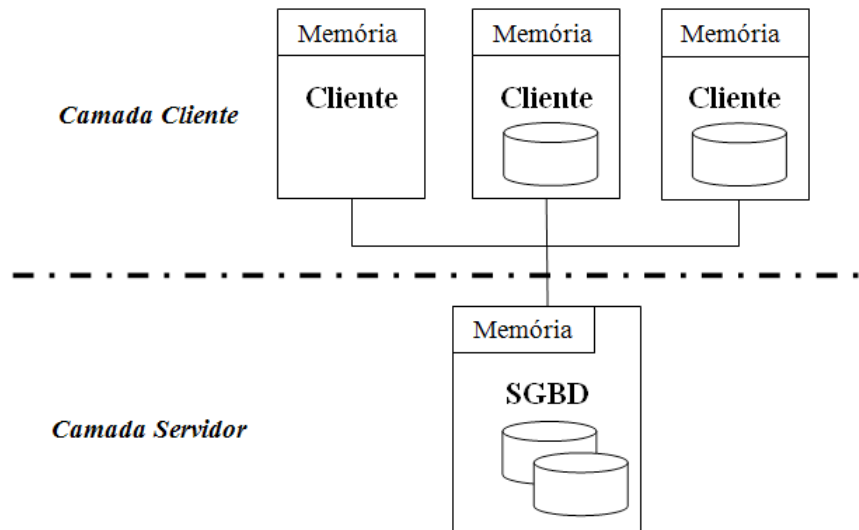


Figura 1.9 – Arquitetura cliente/servidor de duas camadas

Na arquitetura cliente-servidor o programa cliente estabelece uma conexão com o programa servidor para comunicar-se com o SGBD. Após estarem conectados, os sistemas cliente enviam solicitações de manipulação e consulta dos dados, as mesmas são processadas pelo servidor e enviadas de volta para o programa cliente, que processa e apresenta o resultado de acordo com a necessidade.

3.3. Arquitetura Cliente-Servidor de Três Camadas

Com o crescimento da WWW (World Wide Web – Rede de Alcance Mundial – ou simplesmente Web) os papéis na arquitetura cliente-servidor de duas camadas sofreram mudanças que levaram a arquitetura cliente-servidor de duas camadas para a arquitetura cliente-servidor de três camadas, ou arquitetura de três camadas. Na arquitetura de três camadas foi colocada uma camada intermediária entre a máquina cliente e o servidor de banco de dados. Essa camada do meio passou a chamar-se de servidor de aplicação ou servidor Web, dependendo do contexto onde o aplicativo está inserido.

“Esse servidor desempenha um papel intermediário armazenando as regras de negócio [...] que são usadas para acessar os dados do servidor de banco de dados.” [ELMASRI e NAVATHE, 2005]. Checar as credenciais do cliente antes de aceitar a solicitação do mesmo também pode ser incrementado nesse servidor, deixando por sua conta a segurança do banco de dados, liberando o servidor de banco de dados desse processo.

Resumidamente essa arquitetura (figura 1.10) tem três camadas definidas da seguinte maneira:

- Camada de Apresentação (Cliente): Implementa a interface do usuário, também chamada de GUI (Graphics User Interface – Interface Gráfica para o Usuário), podendo incorporar algumas regras de negócios específicas da aplicação, mas seu papel principal é interagir com o usuário fazendo solicitações de dados e apresentando o resultado.
- Camada de Negócio (Servidor de Aplicação): Implementa as funções e regras de negócio, não interagindo diretamente com o usuário, apenas recebendo as solicitações enviada pela camada de apresentação, processando e enviando, ou não, os comandos de banco de dados a camada seguinte. Por fim, recebe e repassa os dados do servidor de banco de dados para a camada de apresentação, não mantendo nenhum dado localmente.
- Camada de Dados (Servidor de Dados): Mantém os dados, servindo como repositório; recebe e executa as solicitações da camada de negócio, enviando de volta a mesma os dados processado. O repasse desses dados para a camada de apresentação é responsabilidade da camada de negócio.

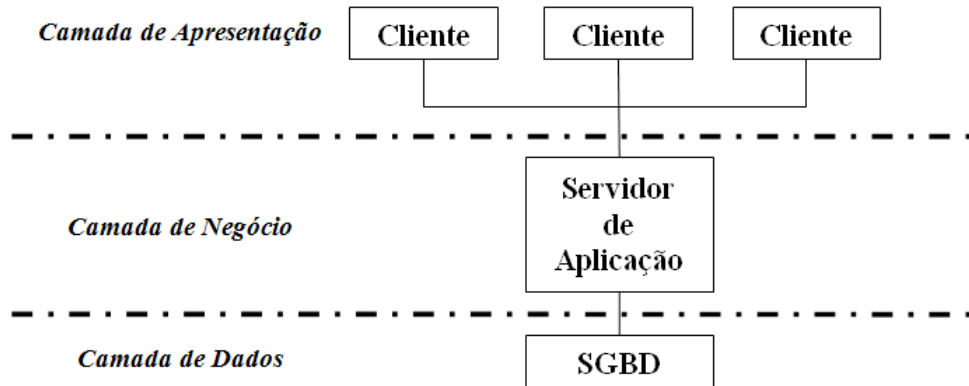


Figura 1.10 – Arquitetura cliente/servidor de três camadas

A arquitetura em três camadas pode parecer igual a arquitetura em duas camadas, mas ao retirar do servidor de dados regras de negócios e de segurança, deixa-se o servidor menos sobrecarregado, pois fica responsável por gerenciar apenas os dados.

Algumas aplicações em duas camadas passavam parte das regras de negócio para o cliente, exigindo um poder computacional maior nessas máquinas. Com a inclusão da camada de negócio deixa-se a camada de apresentação basicamente com a GUI e a camada de dados apenas com o armazenamento e gerenciamento dos dados.

Outra vantagem que pode ser colocada para a arquitetura em três camadas é a escalabilidade da camada intermediária, uma vez que se pode incluir mais de um servidor de aplicação – conectados ao mesmo servidor de dados – e redistribuir o acesso dos clientes entre esses servidores.

3.4. Arquitetura Distribuída

O uso dessa arquitetura traz para o banco de dados não apenas as vantagens da computação distribuída, mas também as dificuldades relacionadas a seu gerenciamento, pois as “funções comuns de gerenciamento do banco de dados [...] não se aplicam, contudo, a esse cenário.” [ELMASRI e NAVATHE, 2005].

Um BDD (Banco de Dados Distribuído) é um conjunto de banco de dados distribuídos através de uma rede de computadores, mas logicamente inter-relacionados, enquanto o SGBDD (Sistema Gerenciador de Banco de dados Distribuído) não apenas gerencia o BDD, mas também torna a distribuição e transações transparentes para o usuário (figura 1.11).

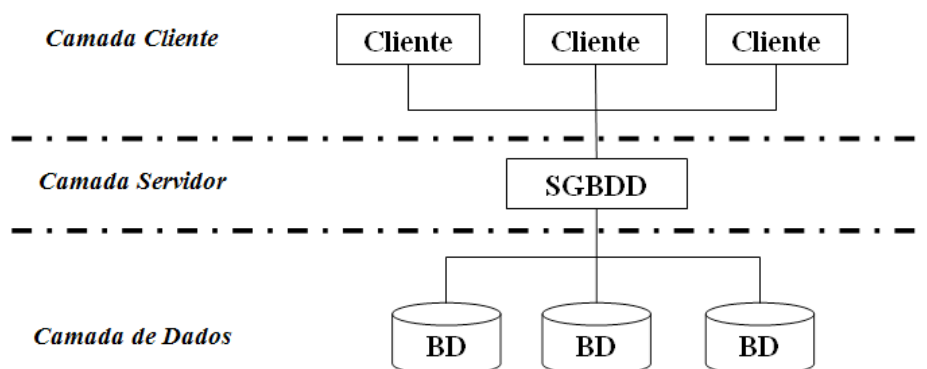


Figura 1.11 – Arquitetura distribuída

Existem dois tipos de banco de dados distribuídos, os (i) homogêneos que são compostos por um único tipo de banco de dados e os (ii) heterogêneos que são compostos por mais de um tipo de banco de dados.

Num banco de dados distribuídos, os dados podem estar replicados ou fragmentados. Na replicação, é criada uma cópia de cada dado em bases diferentes, deixando as bases com os dados iguais. Na fragmentação, os dados são divididos entre bases diferentes.

3.4.1. Vantagens

Alguns dos motivos que levaram ao desenvolvimento dessa arquitetura (BDD) foram a (i) descentralização dos dados, aumentando o poder computacional de processamento; a (ii) fragmentação dos dados levando em consideração a estrutura organizacional, persistindo os dados no local desejado (Ex.: Departamento de Compras) aumentando a autonomia local; a (iii) melhoria no desempenho devido a proximidade dos dados, paralelismo e balanceamento entre os servidores de dados; (iv) tolerância a falhas aumentando a disponibilidade dos dados; (v) economia na aquisição de servidores menores a medida que o poder computacional exigido for aumentado; (vi) facilidade de acrescentar ou remover novos servidores.

Podemos citar ainda, o gerenciamento de dados distribuídos com níveis diferentes de transparência (transparência de distribuição ou de rede, transparência de replicação e transparência de fragmentação). Mas [ELMASRI e NAVATHE, 2005] afirma que as transparências incluem “um compromisso entre a facilidade de uso e o custo da sobrecarga de proporcionar a transparência”.

No banco de dados distribuídos os dados ficam armazenados em locais diferentes. Usualmente cada local é gerenciado por um SGBD independente. “A visão clássica de um sistema de banco de dados distribuído é que o sistema deve tornar o impacto da distribuição dos dados **transparentes**” [RAMAKRISHNAN e GEHRKE, 2008].

3.4.2. Desvantagens

O uso do banco de dados distribuído não só tem vantagens, mas traz consigo algumas desvantagens como (i) complexidade exigida para garantir a distribuição de forma transparente para o usuário; (ii) custo maior de implementação devido ao trabalho extra exigido; (iii) planejamento mais difícil devido a fragmentação, alocação e, algumas vezes, a replicação dos dados; (iv) integridade do banco exige alto recurso de rede; (v) exigência de maior segurança tanto nos servidores quanto na infra-estrutura; (vi) inexistência de um padrão que auxilie a conversão dos bancos de dados centralizados para os banco de dados distribuídos; (vii) poucos casos práticos disponíveis para serem analisados.

Para [ELMASRI e NAVATHE, 2005] a obtenção das vantagens do banco de dados distribuído leva ao projeto e a implementação de um sistema gerenciador mais complexo, onde o SGBDD deve prover, além das funcionalidades do SGBD centralizado, (i) rastreamento de dados; (ii) processamento de consultas distribuídas; (iii) gerenciamento de transações distribuídas; (iv) gerenciamento dos dados replicados; (v) recuperação de banco de dados distribuído; (vi) segurança; (vii) gerenciamento do diretório distribuído.

3.4.3. Fragmentação de Dados

O tipo mais simples de fragmentação de banco de dados é a fragmentação de uma relação inteira, ou seja, os dados de uma tabela inteira são colocados num único servidor do BDD. Dessa maneira sempre que se precisar de alguma informação da tabela, o SGBDD irá busca essa informação no servidor que a mantém. Para exemplificar suponha que um BDD de um supermercado tenha cinco servidores de dados e que as tabelas do BDD estão espalhadas como mostrado na tabela 1.2.

Tabela 1.2

EXEMPLO DE FRAGMENTAÇÃO DE UMA RELAÇÃO INTEIRA		
Servidor	Tabela	# Registros
S01	Cliente	10000
S02	Fornecedor	1000
S03	Compra	1000000
S04	Compra_Item	4000000
S05	Estoque	10000

No servidor S01 ficam armazenados todos os dados de todos os clientes, neste caso os dados de dez mil clientes. Já o servidor S02 é responsável por manter os dados dos fornecedores. O servidor S03 armazena os dados

da compra, tais como data da compra, cliente que efetuou a compra, valor da compra, forma de pagamento, etc.

O servidor S04 registra os itens que o cliente adquiriu em cada compra (arroz, feijão, macarrão, etc.), a quantidade de cada item e o valor de pago pelo item. Os itens em estoque e seus dados (descrição do item, quantidade em estoque, valor de compra, valor de venda, etc.) ficam armazenados no servidor S05.

Neste exemplo o servidor S02 vai precisar de um poder computacional bem menor que o servidor S04 para atender as solicitações feitas, uma vez que o número de registros do servidor S02 é bem menor que do servidor S04, causando um desbalanceamento de carga entre os servidores.

Na fragmentação horizontal, as tuplas (registros) são divididas horizontalmente entre os vários servidores. Esse tipo de fragmentação diminui o problema do desbalanceamento de carga entre os servidores como pode ser visto na tabela 1.3. Com a fragmentação horizontal cada registro da tabela fica em um servidor, junto com todos os seus atributos (colunas).

Tabela 1.3

EXEMPLO DE FRAGMENTAÇÃO HORIZONTAL			
Tabela: Cliente			
Servidor	ID_Cliente	Nome_Cliente	Cidade_Cliente
S01	1	Roberto	Juazeiro do Norte
S02	2	Helena	Fortaleza
S03	3	Francisco	Crato
S01	4	Lucas	Barbalha
S02	5	Ylane	Juazeiro do Norte
S03	6	Eduardo	Barbalha
S01	7	Carlos	Fortaleza
S02	8	Vitor	Fortaleza
S03	9	Maria	Crato

A fragmentação vertical divide as tuplas e seus atributos (tabela 1.4), o problema nesse tipo de fragmentação está na obrigatoriedade de incluir a chave primária ou chave candidata em cada parte da fragmentação, para que esses valores possam ser resgatados e unidos quando necessário.

Existe ainda a possibilidade de combinar a fragmentação horizontal com a fragmentação vertical gerando uma fragmentação mista ou híbrida.

3.4.4. Replicação de Dados

A replicação ou redundância de dados é usada para melhorar a disponibilidade dos dados, pois através dela obtém-se um sistema de alta disponibilidade, mantendo o sistema sempre disponível, mesmo em casos de falhas de componentes ou sobrecargas do sistema.

Tabela 1.4

EXEMPLO DE FRAGMENTAÇÃO VERTICAL					
Tabela: Cliente					
Servidor	ID_Cliente	Nome_Cliente	Servidor	ID_Cliente	Cidade_Cliente
S01	1	Roberto	S04	1	Juazeiro do Norte
S02	2	Helena	S05	2	Fortaleza
S03	3	Francisco	S06	3	Crato
S01	4	Lucas	S05	4	Barbalha
S02	5	Ylane	S04	5	Juazeiro do Norte
S03	6	Eduardo	S06	6	Barbalha
S01	7	Carlos	S06	7	Fortaleza
S02	8	Vitor	S04	8	Fortaleza
S03	9	Maria	S05	9	Crato

Na replicação completa os dados armazenados são replicados de maneira inteira em todos os sites do sistema distribuído. “O outro caso extremo da replicação completa envolve possuir **nenhuma replicação** [...] Entre esse dois extremos, temos um amplo espectro de **replicação parcial** dos dados” [ELMASRI e NAVATHE, 2005].

Embora a replicação de dados melhore a disponibilidade e o desempenho do banco de dados, a mesma reduz a velocidade das operações de atualização, uma vez que cada atualização deverá ser replicada em todas as cópias existentes para manter a consistência dos dados redundantes.

Capítulo

2

Modelagem de Dados e Normalização

Objetivo

- Saber transformar a necessidade do armazenamento de dados num projeto eficaz e eficiente de banco de dados passa por algumas etapas. Este capítulo tem o foco em três etapas importantes do projeto de um banco de dados, iniciando com a modelagem conceitual através de uma abordagem prática, passado pela normalização de dados e finalizando com a modelagem lógica do banco de dados.

“Requisitos de um sistema são descrições dos serviços fornecidos pelo sistema e as suas restrições operacionais” [SOMMERVILLE, 2007].

Introdução

Neste capítulo será apresentando o projeto de um aplicativo que utiliza um banco de dados relacional. No decorrer do capítulo serão proporcionadas técnicas para transformar o levantamento de requisitos, feitos pela engenharia de software, num projeto de banco de dados relacional bem sucedido.

1. Sinopse do Projeto

Um mercantil deseja informatizar alguns processos internos. O controle do estoque é de suma importância, pois a gerência precisa constantemente fazer levantamento de inventário de mercadoria.

Os pontos de venda (PDVs) também devem ser automatizados, inclusive passando a ser obrigatório o uso do Emissor de Cupom Fiscal (ECF).

Com o uso do PDV vinculado a ECF passa a ser obrigatório explicitar os itens que cada cliente levou na compra e a forma de pagamento usado pelo mesmo.

As impressões na ECF dos itens adquirido, assim como a atualização do estoque, devem acontecer concomitantemente, ou seja, quando for passado um item no PDV, o mesmo deve ser impresso na ECF e também ser baixado do estoque a quantidade vendida.

No PDV deve ser informada a data da venda, o cliente que efetuou a compra, o valor da compra e a forma de pagamento utilizada pelo cliente. Os itens vinculados ao PDV devem ter a quantidade comprada, o valor unitário de venda e o valor total.

O cadastro do cliente tem que ser informado com nome, endereço, telefones.

“Engenharia de software é uma disciplina de engenharia relacionada a todos os aspectos da produção de software, desde os estágios iniciais de especificação do sistema até sua manutenção, depois que este entrar em operação” [SOMMERVILLE, 2007].

2. Modelo Conceitual

O primeiro elemento que deve ser criado para um projeto de banco de dados é o modelo conceitual, este modelo deve ser de fácil entendimento para o usuário final, logo ele precisa ser um modelo de alto nível (mais próximo da realidade do usuário).

Nesse momento se procura uma descrição precisa dos dados, sendo necessário especificar quais objetos estão presente no projeto e como eles se relacionam. Detalhes sobre como serão implementados os dados ou relacionamentos devem ser omitidos.

O modelo conceitual deve servir como meio de comunicação não ambíguo entre os usuários do sistema e os desenvolvedores do banco de dados. Obrigando, desta forma, o entendimento e atendimento dos requisitos por estes dois grupos, sendo muito mais valioso para os desenvolvedores reconhecer e validar as reais necessidades do usuário.

No modelo conceitual devem está presente as entidades e seus relacionamentos além dos atributos das entidades, e em alguns casos dos relacionamentos. Uma entidade é a representação, no ambiente de banco de dados, de um objeto do mundo real (professor, aluno, etc.) ou conceitual (disciplina, nota, etc.). Toda entidade tem propriedades (tamanho, cor, nome, etc.), essas propriedades são chamadas de atributos. Existem entidades que se conectam com outros, mostrando uma associação entre as mesmas, essas associações são identificadas como relacionamentos.

No decorrer desta seção será mostrado como criar o modelo conceitual do projeto proposto neste capítulo.

2.1. Identificando os Tipos Entidades

Uma entidade é definida por [ROB e CORONEL, 2011] como “algo (uma pessoa, um local, um objeto, um evento) sobre o qual sejam coletados e armazenados dados. Ela representa um tipo particular de objeto no mundo real”, eles concluem afirmando que as “entidades podem ser objetos físicos, como clientes e produtos, mas também abstrações, como rotas de vôo ou apresentações musicais”.

Embora muitos chamem o tipo entidade de entidade, os dois não são a mesma coisa, mas são complementares. “Um tipo entidade define uma coleção (ou conjunto) de entidades que possuem os mesmos atributos. Cada tipo entidade no banco de dados é descrito por seu nome e atributos” [ELMASRI e NAVATHE, 2005]. Fazendo uma analogia a linguagem de programação, podemos dizer que uma entidade está para um tipo entidade, assim como uma variável está para um tipo primitivo.

Para identificar os tipos entidade presentes num texto, o primeiro procedimento é localizar os substantivos que indiquem um objeto como único no mundo real ou conceitual. No texto apresentado na seção 2.1 foram identificados com tipos entidades os substantivos estoque, PDV, item da venda, forma de pagamento e cliente (figura 2.1).

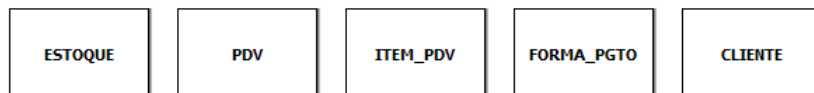


Figura 2.1 – Tipos entidade

Embora mercantil seja um substantivo, ele não apareceu na figura 2.1 porque se o mesmo fosse transformado num tipo entidade, só teria apenas uma única aparição no banco de dados.

Existem outros substantivos no texto (ECF, processos internos, gerência, etc.), mas eles não têm um papel representativo no projeto, suas aparições no texto servem apenas para nos ajuda a entender o projeto.

Outro substantivo descartado foi inventário de mercadoria, pois ele representa uma funcionalidade do software que será criado, devendo ficar claro que esta funcionalidade vai ser criada com base nas informações presentes no tipo entidade ESTOQUE, e não ser um tipo entidade do banco de dados.

2.2. Identificando os Tipos Relacionamento

A identificação dos tipos relacionamento de um projeto é mais simples, pois já tendo descoberto os tipos entidade, o relacionamento entre os mesmo se dá através de verbos ou preposições que os conectem.

Os tipos relacionamento encontrado no texto são apresentados na figura 2.2 como losangos unindo os tipos entidade.

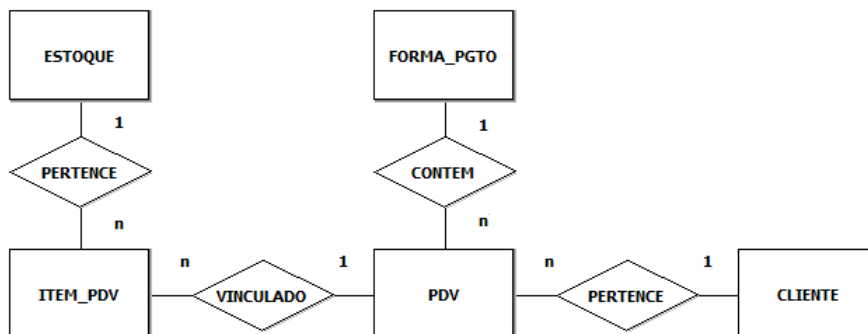


Figura 2.2 – Tipos entidade e tipos relacionamento

Na figura 2.2 também é apresentado a razão de cardinalidade para os relacionamentos. A razão de cardinalidade especifica o número máximo de relacionamentos que uma entidade pode participar.

No tipo relacionamento TORNA-SE, ESTOQUE:ITEM_PDV tem razão 1:N. Em outras palavras, cada entidade estoque pode se relacionar com N entidades item_pdv.

No exemplo todos os tipos relacionamento são de grau binário, ou seja, cada relacionamento suporta a participação de apenas duas entidades. Mas o modelo conceitual permite tipos relacionamento de vários graus, embora seja altamente recomendável tentar utilizar sempre o grau binário.

2.2.1. Identificando os Atributos

Reconhecido os tipos entidade e tipos relacionamento, o próximo passo é identificar os atributos dos tipos entidade. Uma vez que um tipo entidade é a representação de um objeto e todo objeto tem propriedades, devemos representar essas propriedades através dos atributos. O modelo conceitual também permite aos tipos relacionamento terem atributos. Um atributo também pode ser visto como um substantivo que descreve outro substantivo.

A figura 2.3 representa os atributos encontrados na seção 2.1 através de círculos fixados diretamente no tipo entidade.

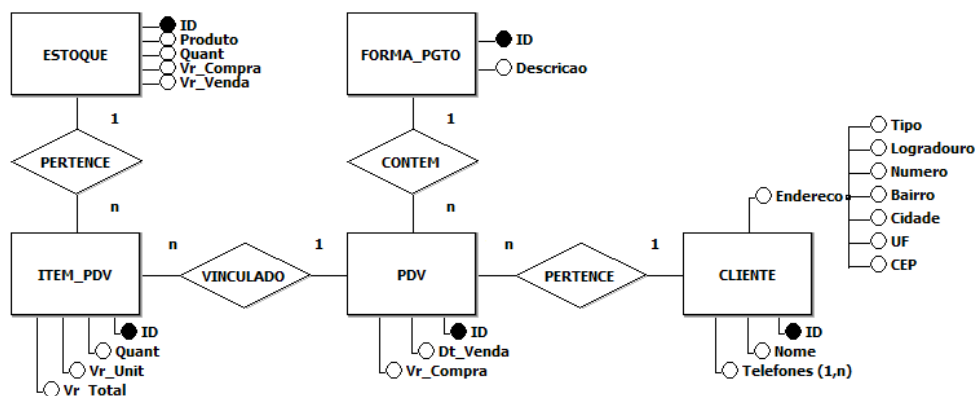
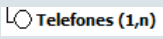
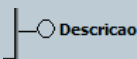
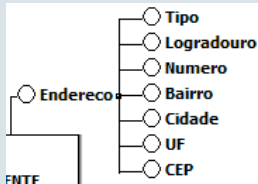
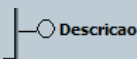

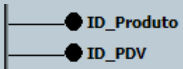


Figura 2.3 – Modelo conceitual do projeto

A tabela 2.1 explana sobre os tipos de atributos representados na figura anterior.

Tabela 2.1

TIPOS DE ATRIBUTOS ENCONTRADOS NA FIGURA 2.3		
Atributo	Exemplo	Descrição
Atributo multivalorado		Atributo formado por um conjunto de valores por entidade.
Atributo monovalorado		Atributo formado por um único valor por entidade. Este atributo é o complemento do atributo multivalorado.
Atributo composto		Atributo que pode ser dividido em partes menores, formando uma hierarquia de atributos.
Atributo simples ou atômico		Atributo que não pode ser dividido em partes menores. Este atributo é o complemento do atributo composto.
Atributo chave simples		Também chamado de restrição de unicidade ou chave primária simples. Este atributo não pode ter o valor nulo, nem ter duas entidades com o mesmo valor.
Atributo chave composto		Semelhante ao atributo chave simples, mas nesse caso são usados dois ou mais atributos para definir a restrição de unicidade. Também pode ser chamado de chave primária composta.

2.3. Normalização

Com o modelo conceitual pronto, o passo seguinte é transformá-lo no modelo lógico, mas antes devemos ter o conhecimento das formas normais. Esse conhecimento é necessário, pois o modelo conceitual não se preocupa com a implementação, enquanto o modelo lógico apresenta uma visão abstrata apropriada a equipe de desenvolvimento.

A normalização procura simplificar a maneira como os dados serão armazenados no banco de dados para conseguir mais eficiência. Neste contexto a palavra “eficiência” não se refere melhorar o desempenho do banco de dados ou facilitar o processo de consulta. A eficiência procurada aqui se refere a diminuição da complexidade da estrutura lógica do banco de dados.

A normalização é o processo de análise efetuado sobre esquemas relacionais para conseguir características desejáveis, tais como a minimização de redundância e, conseqüentemente, a redução de anomalias de inserção, atualização e exclusão.

A redundância de dados acontece quando “uma determinada informação está representada no sistema em computador várias vezes” [HEUSER, 2001]. Um exemplo de normalização apresentado por [ELMASRI e NAVATHE, 2005] é o de um banco de dados de uma universidade, onde dois grupos de usuários (secretaria e contabilidade) mantêm arquivos independentes com os dados dos alunos. “A contabilidade também guarda os dados de matrícula e as informações relacionadas a pagamentos, enquanto a secretaria mantém o controle dos cursos e notas dos alunos” [ELMASRI e NAVATHE, 2005].

2.3.1. Forma Normal

Uma forma normal é uma regra que deve ser seguida para que uma tabela seja bem avaliada. A forma normal sujeita o esquema de relação a uma cadeia de avaliação para garantir que ele satisfaz a forma normal. Esse processo de avaliação segue o estilo top-down, onde cada relação é avaliada sob os critérios das formas normais.

2.3.1.1. Primeira Forma Normal (1FN)

Uma tabela está na 1FN se não possuir atributo multivalorado ou atributo composto, esse procedimento elimina tabelas aninhadas. A figura 2.4 mostra uma tabela que não atende a 1FN, pois temos um atributo multivalorado (Telefone) e um atributo composto (Endereco).

ID	Nome	Telefone	Endereco
1	Antonio	(85) 3211-0000 (85) 3212-0000 (85) 9988-0000	Rua Padre Cicero, 999 – Aldeota – Fortaleza – CE
2	Joana	(88) 3566-0000 (88) 9977-0000	Rua São Paulo, 355 – Matriz – Juazeiro do Norte – CE
3	Maria	(81) 8881-0000	Av. Caxangá, 1200 – Centro – Recife – PE
4	José	(88) 3521-0000	Rua Dom Manuel, 208 – Centro – Crato – CE

Figura 2.4 – Tabela fora da 1FN

Para resolver o problema do atributo multivalorado, deve-se criar uma nova tabela com o atributo multivalorado (figura 2.5), essa nova tabela deve se relacione com a tabela.

<u>ID Telefone</u>	Telefone	ID
1	(85) 3211-0000	1
2	(85) 3212-0000	1
3	(85) 9988-0000	1
4	(88) 3566-0000	2
5	(88) 9977-0000	2
6	(81) 8881-0000	3
7	(88) 3521-0000	4

Figura 2.5 – Tabela criada com base no campo multivalorado

O problema do atributo composto é mais simples, os atributos base devem ser inseridos direto na tabela, eliminando-se o atributo compostos (figura 2.6).

<u>ID</u>	Nome	Tipo	Logradouro	Nro	Bairro	Cidade	UF
1	Antonio	Rua	Padre Cicero	999	Aldeota	Fortaleza	CE
2	Joana	Rua	São Paulo	355	Matriz	Juazeiro do Norte	CE
3	Maria	Avenida	Caxangá	1200	Centro	Recife	PE
4	José	Rua	Dom Manuel	208	Centro	Crato	CE

Figura 2.6 – Tabela na 1FN

A figura 2.7 mostra o modelo conceitual do exemplo apresentado para a 1FN.

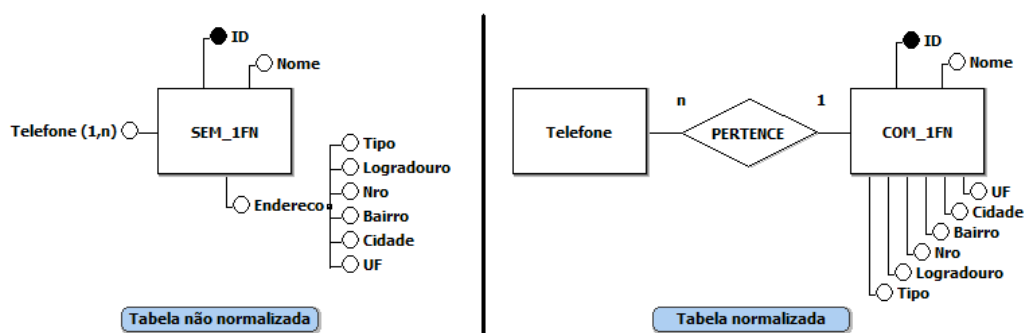


Figura 2.7 – Modelo conceitual da 1FN

2.3.1.2. Segunda Forma Normal (2FN)

Uma tabela está na 2FN se estiver na 1FN e não possuir dependência funcional parcial. Uma dependência parcial ocorre quando os atributos não chave não dependem de toda chave primária composta (Ler Tabela 2.1 – Atributo

chave simples e Atributo chave composto). A figura 2.8 mostra uma tabela que não atende a 2FN.

<u>ID</u>	<u>ID Produto</u>	Descricao	Vr_Unit	Qty	Vr_Total
1	1234	Camiseta	30,00	2	60,00
1	4321	Calça	120,00	1	120,00
2	1234	Camiseta	30,00	3	90,00
2	4321	Calça	120,00	2	240,00

Figura 2.8 – Tabela fora da 2FN

Neste exemplo a **chave primária** é composta por dois atributos (ID, ID_Produto). Dois **atributos não chave** (Descricao, Vr_Unit) têm dependência funcional parcial com a **chave primária**. Explanando de outra maneira, os dois **atributos não chave** mencionados têm seus valores diretamente relacionados com o atributo ID_Produto. A figura 2.9 apresenta a tabela já atendendo a 2FN.

<u>ID Produto</u>	Descricao	Vr_Unit
1234	Camiseta	30,00
4321	Calça	120,00

<u>ID</u>	<u>ID Produto</u>	Qty	Vr_Total
1	1234	2	60,00
1	4321	1	120,00
2	1234	3	90,00
2	4321	2	240,00

Figura 2.9 – Tabela na 2FN

A figura 2.10 mostra o modelo conceitual do exemplo apresentado para a 2FN.

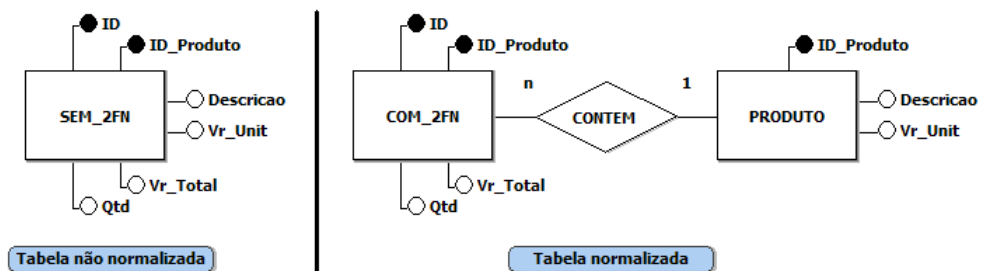


Figura 2.10 – Modelo conceitual da 2FN

2.3.1.3. Terceira Forma Normal (3FN)

Uma tabela está na 3FN se estiver na 2FN e não possuir nenhuma dependência funcional transitiva. Uma dependência transitiva ocorre quando um **atributo não chave** depende de outro **atributo não chave**. A figura 2.11 mostra uma tabela que não atende a 3FN.

<u>ID</u>	Nome	ID_Cargo	Cargo	Salario
1	Antonio	1	Engenheiro	7.000,00
2	Joana	2	Médico	10.000,00
3	Maria	3	Advogado	50.000,00
4	José	1	Engenheiro	7.000,00

Figura 2.11 – Tabela fora da 3FN

Os **atributos não chave** Cargo e Salario têm dependência funcional transitiva com o **atributo não chave** ID_Cargo. A solução é semelhante a da 2FN, ou seja, cria-se uma nova tabela para solucionar a dependência funcional (figura 2.12).

<u>ID Cargo</u>	Cargo	Salario
1	Engenheiro	7.000,00
2	Médico	10.000,00
3	Advogado	50.000,00

<u>ID</u>	Nome	ID_Cargo
1	Antonio	1
2	Joana	2
3	Maria	3
4	José	1

Figura 2.12 – Tabela na 3FN

A figura 2.13 mostra o modelo conceitual do exemplo apresentado para a 3FN.

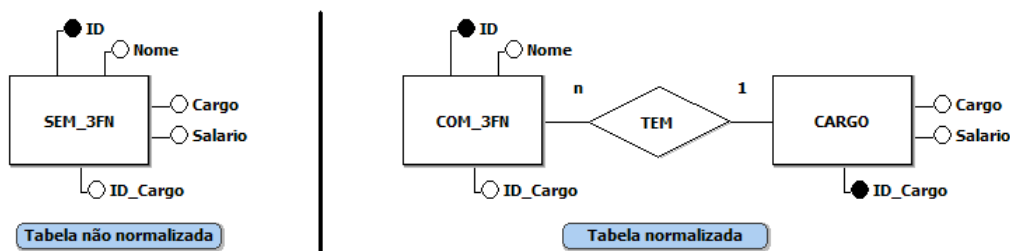


Figura 2.13 – Modelo conceitual da 3FN

Na maioria dos projetos, quando se chega a 3FN o esquema do banco de dados já está com a normalização satisfatória. Mas em alguns casos, para atingir uma normalização aceitável é necessário executar a 4FN e 5FN.

2.3.1.4. Quarta Forma Normal (4FN)

Uma tabela está na 4FN se estiver na 3FN e não existir dependência funcional multivalorada. Uma dependência multivalorada ocorre quando dois ou mais atributos multivalorados dependem de um atributo chave. A figura 2.14 mostra uma tabela que não atende a 4FN.

<u>ID</u>	Filme	Ator	Produtor
1	Ben-Hur	Charlton	Karl
		Jack	William
			Stephen
2	O Sétimo Selo	Gunnar	Ingmar
		Bengt	

Figura 2.14 – Tabela fora da 4FN

Os **atributos multivalorados** Ator e Produtor têm dependência funcional multivalorada com o **atributo chave** ID. Neste caso, cada **atributo multivalorado** se transformará numa tabela independente da tabela original (figura 2.15).

<u>ID</u>	Filme
1	Ben-Hur
2	O Sétimo Selo

<u>ID Ator</u>	Ator
1	Charlton
2	Jack
3	Gunnar
4	Bengt

<u>ID Prod</u>	Produtor
1	Karl
2	William
3	Stephen
4	Ingmar

Figura 2.15 – Tabelas derivadas da figura 2.14

Cada tabela derivada dos **atributos multivalorados** deve se relacionar com a tabela original através de uma tabela intermediária (figura 2.16). As tabelas intermediárias têm o **atributo chave** da tabela original (ID) e o **atributo chave** das tabelas derivadas (ID_Atör, ID_Prod). O produtor do filme produziu.

<u>ID</u>	<u>ID Ator</u>
1	1
1	2
2	3
2	4

<u>ID</u>	<u>ID Prod</u>
1	1
1	2
1	3
2	4

Figura 2.16 – Tabelas intermediárias

A figura 2.17 mostra o modelo conceitual do exemplo apresentado para a 4FN.

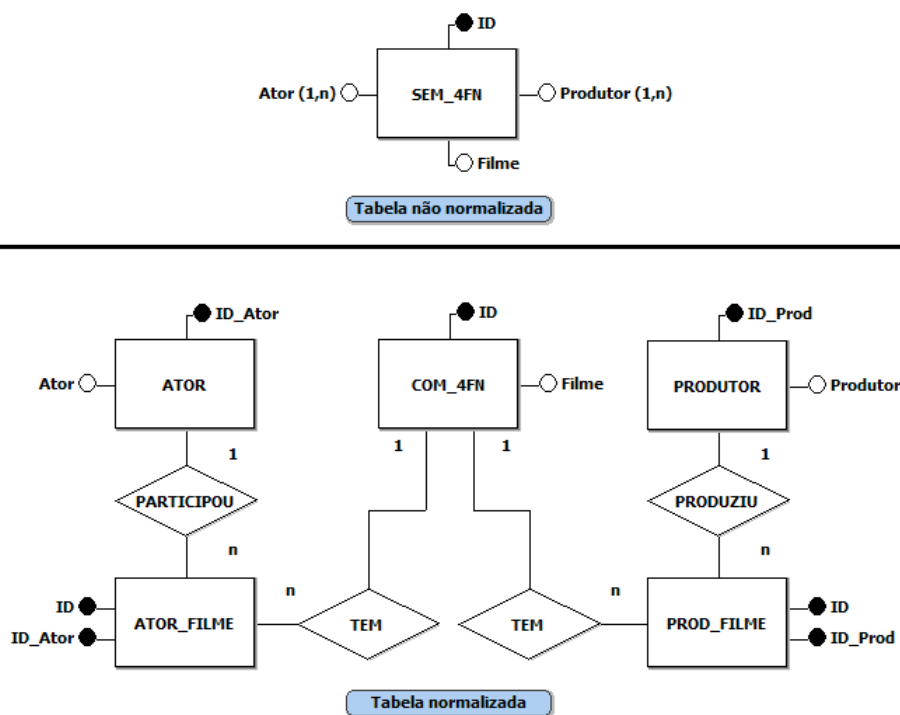


Figura 2.17 – Modelo conceitual da 4FN

2.3.1.5. Quinta Forma Normal (5FN)

Uma tabela está na 5FN se estiver na 4FN e não existir dependência funcional cíclica. Uma dependência cíclica ocorre quando um atributo **X** depende do atributo **Y**, o atributo **Y** depende do atributo **Z** e o atributo **Z** depende do atributo **X**. A figura 2.18 mostra uma tabela que não atende a 5FN.

<u>ID Prof</u>	Professor	<u>ID Disc</u>	Disciplina	<u>ID Apost</u>	Apostila
1	Tadeu	11	Banco de Dados	111	Tutorial Banco de Dados
2	Robério	22	POO	222	Tutorial POO
3	José Maria	33	Marketing	333	Tutorial Marketing

Figura 2.18 – Tabela fora da 5FN

A tabela apresentada na figura 2.18 poderia mostrar que o professor Tadeu ministra a disciplina Banco de Dados, e que só poderá ministrar essa disciplina se utilizar a apostila Tutorial Banco de Dados. Desta maneira é criada uma dependência cíclica entre os atributos Professor, Disciplina e Apostila. Para evitar essa dependência cíclica, devem-se criar novas tabelas que relacionem esses atributos de forma binária (figura 2.19).

<u>ID Prof</u>	Professor	<u>ID Disc</u>	Disciplina	<u>ID Apost</u>	Apostila
1	Tadeu	11	Banco de Dados	111	Tutorial Banco de Dados
2	Robério	22	POO	222	Tutorial POO
3	José Maria	33	Marketing	333	Tutorial Marketing

<u>ID Disc</u>	<u>ID Apost</u>
11	111
22	222
33	333

<u>ID Prof</u>	<u>ID Disc</u>
1	11
2	22
3	33

<u>ID Prof</u>	<u>ID Apost</u>
1	111
2	222
3	333

Figura 2.19 – Tabelas derivadas da figura 2.18

A figura 2.20 mostra o modelo conceitual do exemplo apresentado para a 5FN.

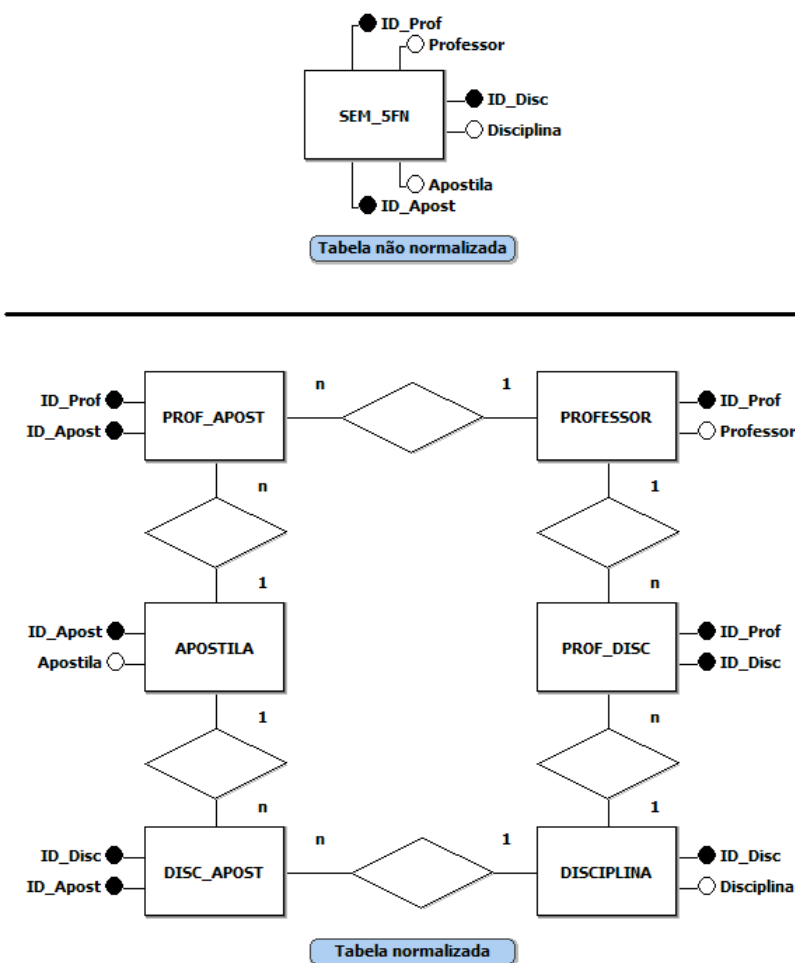


Figura 2.20 – Modelo conceitual da 5FN

2.3.2. Desnormalização

A desnormalização é um assunto pouco mencionado na literatura por ser controverso e não ser uma técnica unânime. Mas em certos casos podem requerer a redundância de parte dos dados, ou que **atributos não relacionados** sejam combinados em **tipos entidade simples**.

Um caso em particular que requer a desnormalização é a necessidade de **manter um histórico** sobre os dados armazenados. Um exemplo dessa necessidade é a obrigatoriedade de manter os dados dos documentos fiscais emitidos sem alteração, após terem sido consolidados. Para ficar mais claro será apresentado um cenário que reflete melhor esse exemplo:

1. A distribuidora ABCD emite uma nota fiscal eletrônica (NF-e) para o cliente XYZ, no ato da emissão da NF-e o cliente XYZ morava na RUA J50, tendo efetuado a compra de 100 refrigerantes no valor unitário de R\$ 2,95.

No capítulo 5 será apresentado o conceito e exemplos práticos sobre junções de tabelas.

2. Um ano após a emissão da NF-e, a Secretaria da Fazenda solicita o reenvio dos dados da mesma. A distribuidora ABCD sabe que os dados que serão enviados devem ser idênticos aos enviados um ano antes no ato da emissão.
3. Atualmente o cliente XYZ está morando na RUA WASHINGTON SOARES, para piorar a circunstância o refrigerante que o cliente comprou já mudou de preço algumas vezes.
4. A distribuidora ABCD não conseguirá atender a solicitação da Secretaria da Fazenda, pois para evitar redundância de dados, o banco de dados foi totalmente normalizado.

Outro caso, ainda mais controverso, é a desnormalização para evitar consultas complexas que são requeridas constantemente. Entenda-se com consultas complexas aquelas que fazem uso de junções entre duas ou mais tabelas para chegar ao resultado desejado.

Mas desnormalizar para ganhar desempenho talvez seja o mais controverso argumento para desnormalizar um banco de dados, pois é muito difícil comprovar que o ganho de desempenho com a desnormalização é significativo.

2.4. Modelo Lógico

O modelo lógico apresenta uma visão abstrata apropriada a equipe de desenvolvimento. Um modelo lógico eficiente tem que está normalizado e ter as chaves estrangeiras criadas corretamente.

Observação

A partir deste ponto (i) **tipo entidade** será citado como **tabela**; (ii) **entidade** será citada como **registro**; e (iii) **atributo** será citado como **coluna** ou **campo**.

2.4.1. Chave Estrangeira

Uma chave estrangeira é uma ou mais **colunas** de uma **tabela** cujos valores devem, fundamentalmente, está presente como chave primária de outra **tabela**. Recordando que a chave primária (ou atributo chave) é uma ou mais colunas cujos valores tornam um **registro** como único na **tabela**.

No modelo lógico as chaves estrangeiras substituem os **tipos relacionamento** do modelo conceitual, mas essa substituição atende a determinadas regras baseadas na cardinalidade existente no modelo conceitual:

Cardinalidade 1:1 – Neste tipo de relacionamento, na maioria das vezes, a chave estrangeira pode ser criada em qualquer uma das tabelas. A figura 2.21 mostra o modelo conceitual e seu similar no modelo lógico para a cardinalidade 1:1.

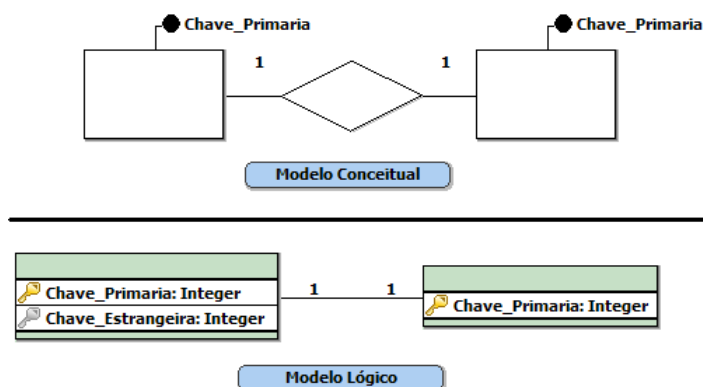


Figura 2.21 – Cardinalidade 1:1

Cardinalidade 1:N – Neste tipo de relacionamento a chave estrangeira deve ser criada na tabela que tem a cardinalidade N (figura 2.22).

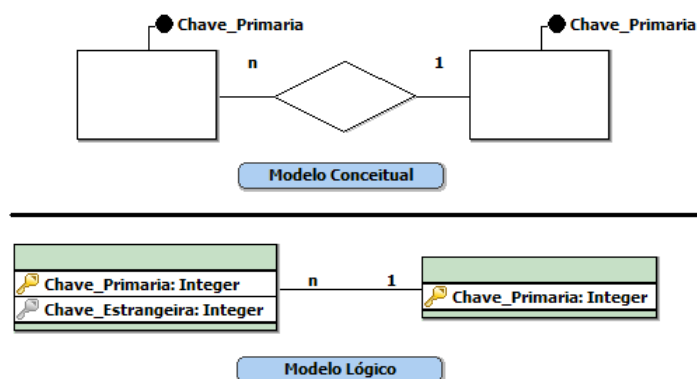


Figura 2.22 – Cardinalidade 1:N

Cardinalidade N:N – Neste tipo de relacionamento, deve-se criar uma nova tabela e inserir nela a chave estrangeira referente as tabelas envolvidas no relacionamento. As tabelas antigas passam a se relacionar através da nova tabela através de um cardinalidade 1:N (figura 2.23).

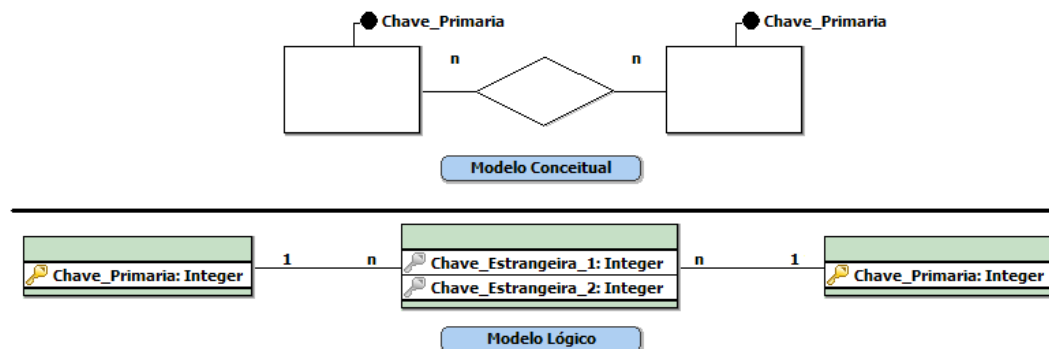


Figura 2.23 – Cardinalidade N:N

2.4.2. Criação do Modelo Lógico

O modelo lógico criado será baseado no modelo conceitual do projeto, apresentado na figura 2.3. Como o modelo lógico deve estar normalizado, deve-se verificar se a figura 2.3 precisa ser normalizada. No modelo conceitual da figura citada, só é necessário normalizar tabela CLIENTE, devido o mesmo apresentar uma coluna composta (Endereco) e outra multivalorada (Telefone). Após normalizar, o modelo conceitual ficará como mostra a figura 2.24.

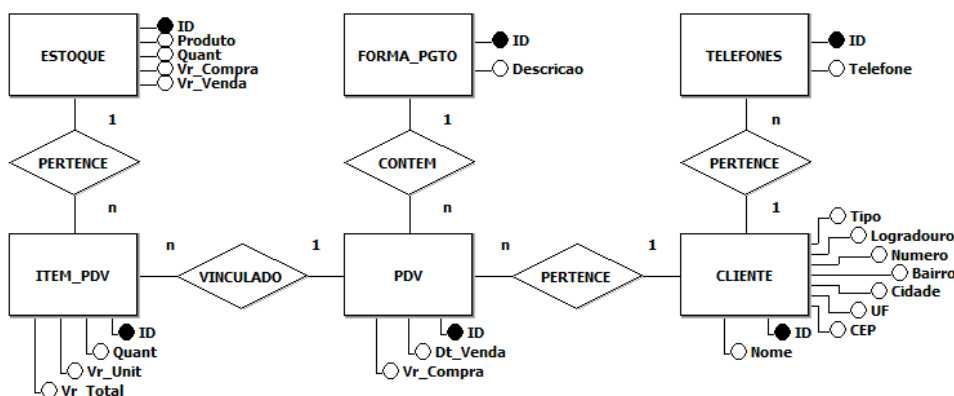


Figura 2.24 – Modelo conceitual do projeto normalizado

Os tipos de dados mais comuns utilizados nos bancos de dados serão apresentados no capítulo 4.

Tendo como base o modelo conceitual mostrado na figura 2.24 é possível criar o modelo lógico (figura 2.25), transformando os relacionamentos em chaves estrangeiras, de acordo com as regras apresentadas anteriormente.

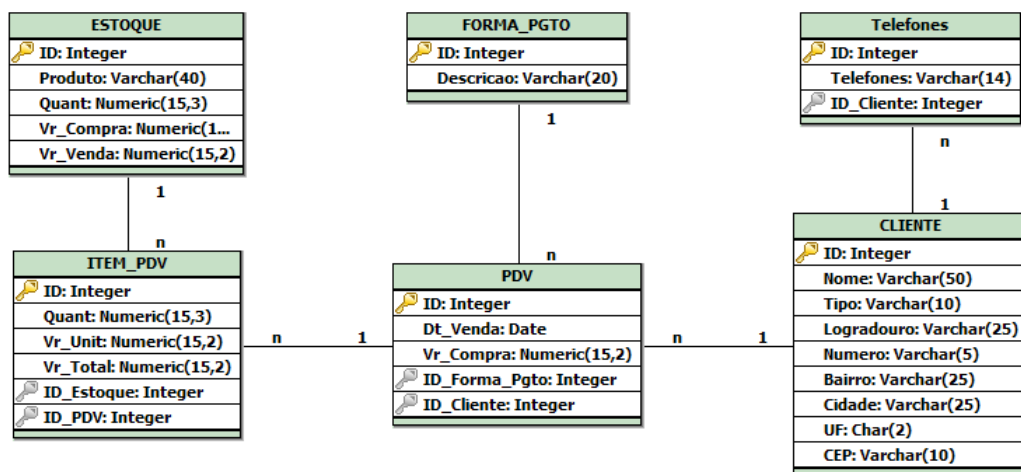


Figura 2.25 – Modelo lógico do projeto

Além de transformando os relacionamentos em chaves estrangeiras, o modelo lógico traz uma informação a mais, o tipo de dado que pode ser armazenado com cada coluna da tabela.

Capítulo

3

PostgreSQL e Modelagem Física

Objetivo

- Um projeto de um banco de dados é finalizado com a escolha do banco de dados que será usado e com a criação do modelo físico. O modelo físico está diretamente relacionado ao banco de dados escolhido, uma vez que o mesmo representa a forma como os dados serão armazenados fisicamente. O capítulo abordará o banco de dados PostgreSQL e como criar o modelo físico baseado no mesmo.

Introdução

O capítulo anterior apresentou o projeto de um aplicativo que utiliza um banco de dados relacional. Foram utilizadas técnicas para a criação do modelo conceitual e do modelo lógico do projeto baseados nos requisitos levantados.

Neste capítulo iremos explicar como criar o modelo físico baseado no modelo lógico apresentado no capítulo 2. O modelo físico é criado via SQL (Structured Query Language – Linguagem Estruturada de Consulta) e executado diretamente no SGBD, devido a isto, o capítulo inicia mostrando como instalar o SGBD PostgreSQL no sistema operacional Linux sobre a distribuição Ubuntu.

Open source refere-se aos software com código aberto e foi criado pela Open Source Initiative (OSI). Acesse <http://opensource.org/> para maiores informações.

1. PostgreSQL

O PostgreSQL é um SGBD proveniente do POSTGRES que foi escrito na Universidade da Califórnia em Berkely. A primeira versão de demonstração do POSTGRES tornou-se operacional em 1987, em 1994 passou a se chamar Postgres95 e em 1996 recebeu o nome PostgreSQL, sendo o nome usado até hoje.

O PostgreSQL é um projeto open source coordenado pelo PostgreSQL Global Development Group, tendo seu desenvolvimento sido feito por um grupo de desenvolvedores distribuídos pelo mundo, em sua maioria, voluntários. É considerado “atualmente o mais avançado banco de dados de código aberto disponível em qualquer lugar.” [POSTGRESQL, 2011].

1.1. Instalando o PostgreSQL

Essa subseção mostrará como instalar o PostgreSQL 8.4 no sistema operacional Linux sobre a distribuição Ubuntu 10.4.

O primeiro passo a ser efetuado é fazer o download do instalador do PostgreSQL. O arquivo que será utilizado é o **postgresql-8.4.8-1-linux.bin**, baixado do endereço <http://www.postgresql.org/download/>.

Acesse http://www.cassic.com.br/carregar/tutoriais/SERVL_TIPACESS para maiores informações sobre o comando **chmod**.

Após o download deve-se entrar no console (terminal) do Ubuntu e entrar no diretório onde se encontra o arquivo de instalação (Ex.: **cd /home/linuxubuntu-vb/Downloads**). As permissões do arquivo devem ser mudadas através do comando **chmod 755 postgresql-8.4.8-1-linux.bin**.

Ainda no console do Ubuntu o comando **sudo ./postgresql-8.4.8-1-linux.bin** deve ser executado para iniciar a instalação. Deste ponto por diante a instalação será no modo visual. Todos os passos seguintes serão apresentados em tópicos, a mudança entre as janelas acontecerá através do botão Next. Serão mantidos os dados padrões apresentados nas janelas, a menos que seja solicitada a sua alteração.

1. A primeira janela exposta pelo instalador é a de boas-vindas (figura 3.1).

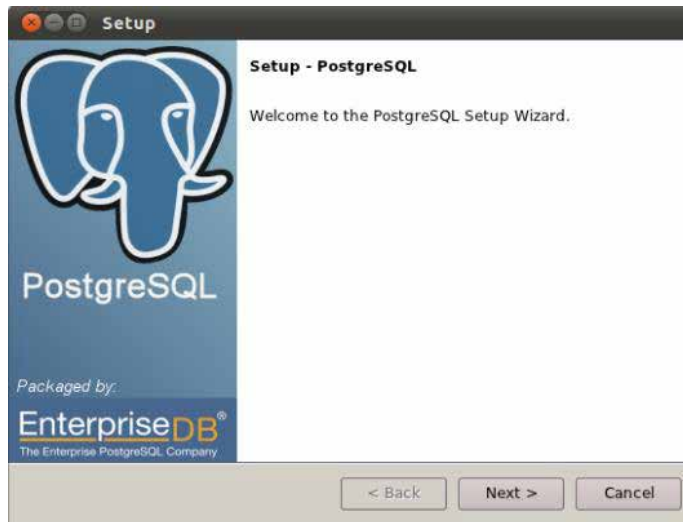


Figura 3.1 – Janela de boas-vindas.

2. Na janela seguinte (figura 3.2) é solicitado o diretório onde o PostgreSQL será instalado.

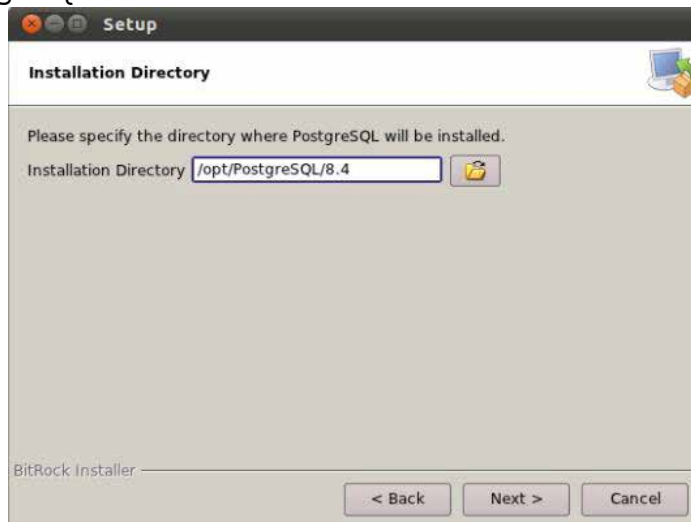


Figura 3.2 – Diretório de instalação.

3. Em seguida (figura 3.3) é solicitado o diretório onde os dados serão mantidos.



Figura 3.3 – Diretório de dados.

4. A senha do banco de dados deve ser informada e confirmada (figura 3.4). A senha usada foi postgres, que é o mesmo nome do super-usuário.

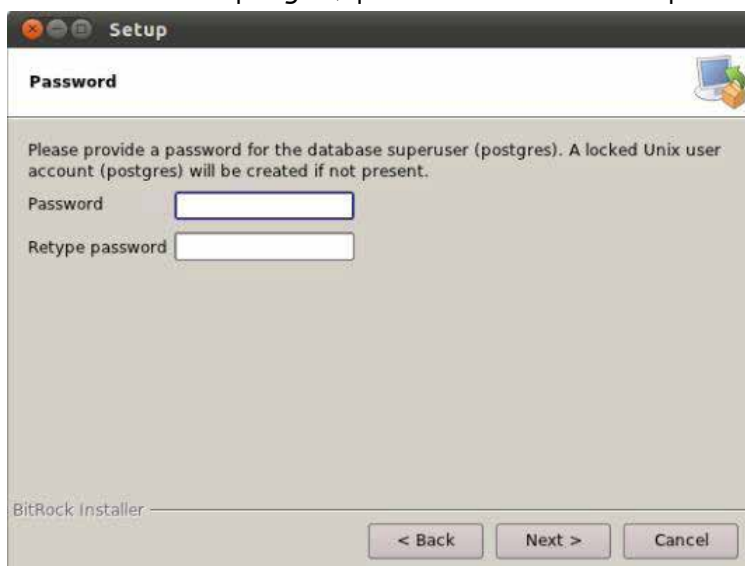


Figura 3.4 – Criação da senha.

5. A porta utilizada pelo servidor para atender as solicitações deve ser informada como mostrado figura 3.5.



Figura 3.5 – Porta usada pelo servidor.

6. Em opções avançadas é solicitado o local que será usado num provável cluster de banco de dados. Também é solicitado a confirmação da instalação da pl/pgsql no template1 do banco de dados. Deixe as opções como mostradas na figura 3.6.

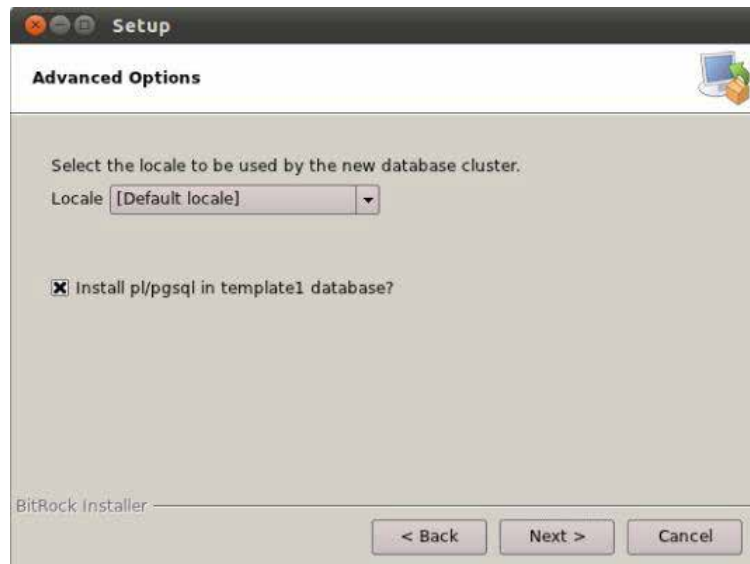


Figura 3.6 – Opções avançadas.

Para o PostgreSQL um *cluster* de banco de dados é um conjunto de bancos de dados gerenciada por uma única instância de um servidor de banco de dados.

7. Na figura 3.7 o instalador informa que está pronto para iniciar instalação do PostgreSQL. Clicar no botão Next.



Figura 3.7 – Instalação pronta para iniciar.

8. Após completar a instalação do PostgreSQL, o instalador pergunta se é desejável baixar e instalar ferramentas adicionais, drivers e aplicações complementares através do Stack Builder (figura 3.8). Com a opção marcada, clicar no botão Finish.

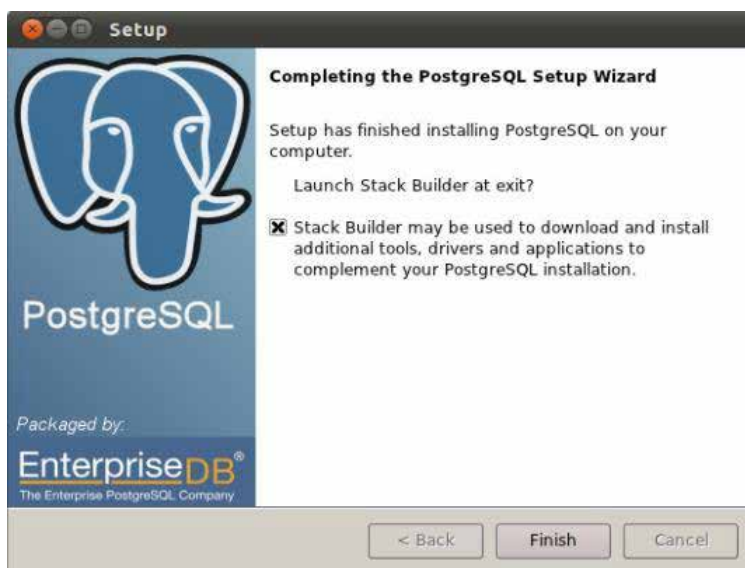


Figura 3.8 – Instalação concluída.

9. Na janela de boas-vindas do Stack Builder informe a opção PostgreSQL 8.4 on port 5432 (figura 3.9).



Figura 3.9 – Janela de boas-vindas do Stack Builder.

10. A janela seguinte do Stack Builder solicita os aplicativos que deverão ser instalados. Marcar as opções apresentadas na figura 3.10.

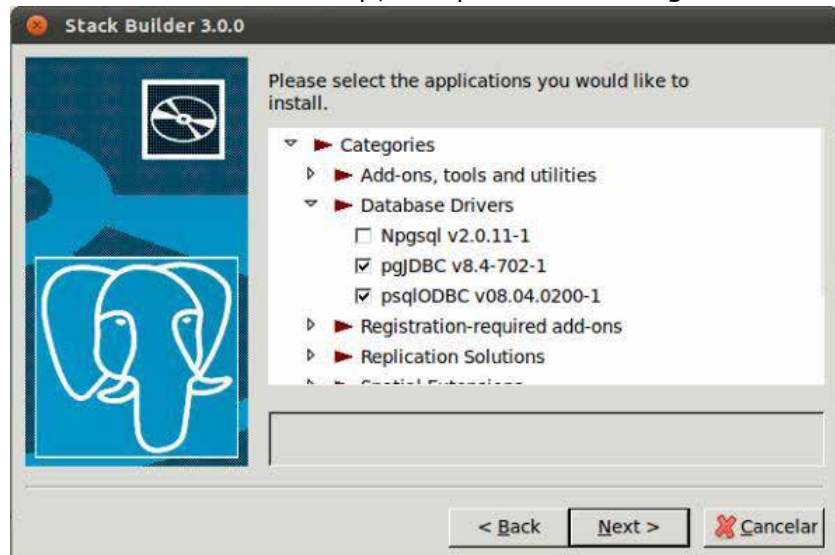


Figura 3.10 – Aplicativos a serem instalados.

11. Antes de iniciar o download e instalação dos aplicativos, o Stack Builder pede a revisão das escolhas feitas na janela anterior, assim como o diretório que será usado para o download (figura 3.11).

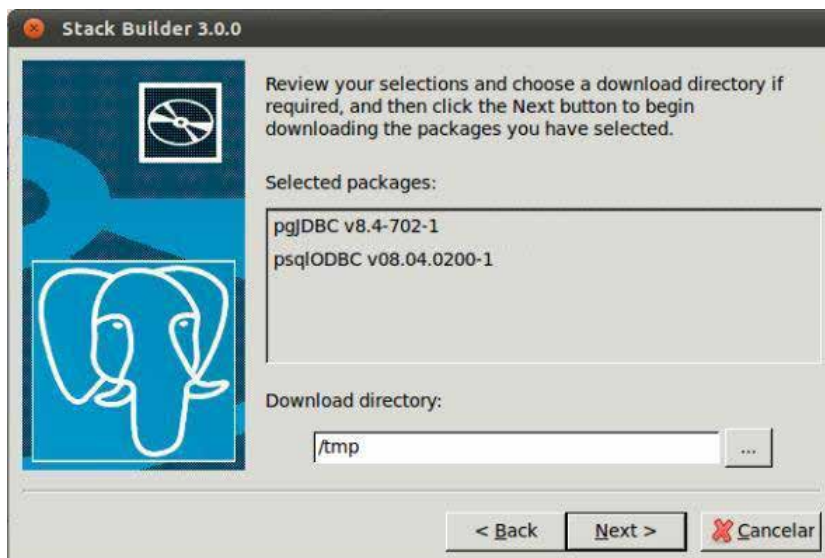


Figura 3.11 – Confirmação de instalação.

12. Após efetuar o download de todos os arquivos de instalação, o Stack Builder solicita um clique no botão Next para iniciar as instalações (figura 3.12).



Figura 3.12 – Downloads efetuado com sucesso.

13. Depois de ter instalado todos os complementos, o botão Finish deve ser acionado (figura 3.13).



Figura 3.13 – Instalação concluída.

Um **front-end** é uma interface responsável por coletar os dados de entrada inseridos pelo usuário, efetuar um pré-processamento e enviá-los ao seu destino final (*back-end*), captura a resposta e apresenta ao usuário de forma inteligível.

O próximo capítulo é dedicado exclusivamente a SQL, onde será possível conhecer vários comandos da SQL.


2. Modelo Físico

Após instalar o PostgreSQL, deve-se utilizar a SQL para criar o modelo físico do banco de dados. Nesta seção será usado o front-end pgAdmin para automatizar o uso da SQL.

2.1. pgAdmin

O pgAdmin é o front-end usado pelo PostgreSQL. Com ele é possível manipular o banco de dados de várias maneiras. Mas como afirmado anteriormente, neste momento o pgAdmin será usado para automatizar a criação do modelo físico.

Para iniciar o pgAdmin deve-se selecionar o menu **Aplicativos** do Linux Ubuntu e na sequência escolher a opção **PostgreSQL 8.4** e em seguida **pgAdmin III**. Esse procedimento irá abrir o pgAdmin (figura 3.14).

O pgAdmin permite ao usuário criar várias conexões com um ou mais servidores. Para criar uma conexão com o servidor local deve-se clicar no botão  e preencher como mostrado na figura 3.15 e clique no botão **OK**.

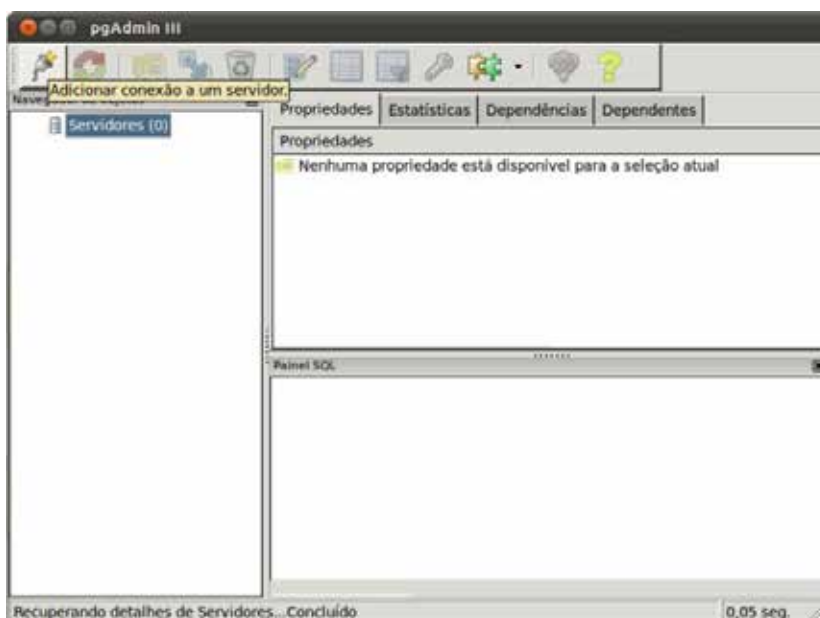


Figura 3.14 – Janela inicial do pgAdmin.

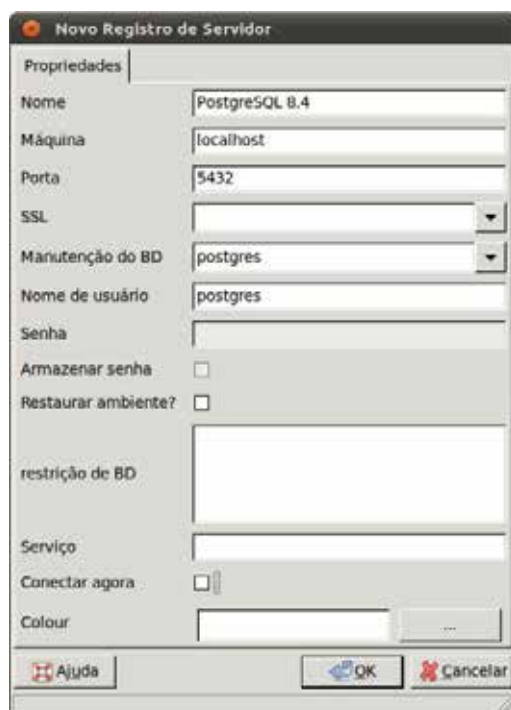


Figura 3.15 – Configuração da nova conexão com o servidor.

Para conectar-se com o servidor, é necessário dá um clique duplo sobre a conexão configurada na figura 3.15, que está no Navegador de objetos do pgAdmin (lado esquerdo). Na janela Conectar ao Servidor (figura 3.16) deve ser informado a senha criada na instalação do PostgreSQL (a senha usada foi **postgres**).

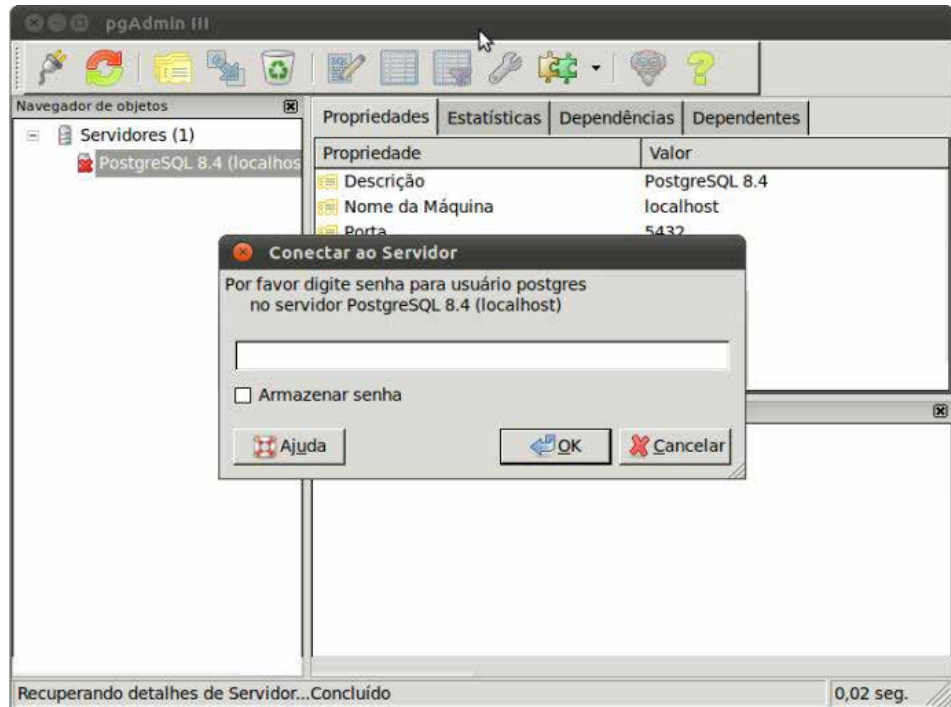


Figura 3.16 – Abrindo conexão com servidor.

2.1.1. Criando a Tabela ESTOQUE

No Navegador de objetos abra o objeto Banco de Dados, em seguida os objetos postgres, Esquemas e public para ter acesso ao objeto Tabelas (figura 3.17).

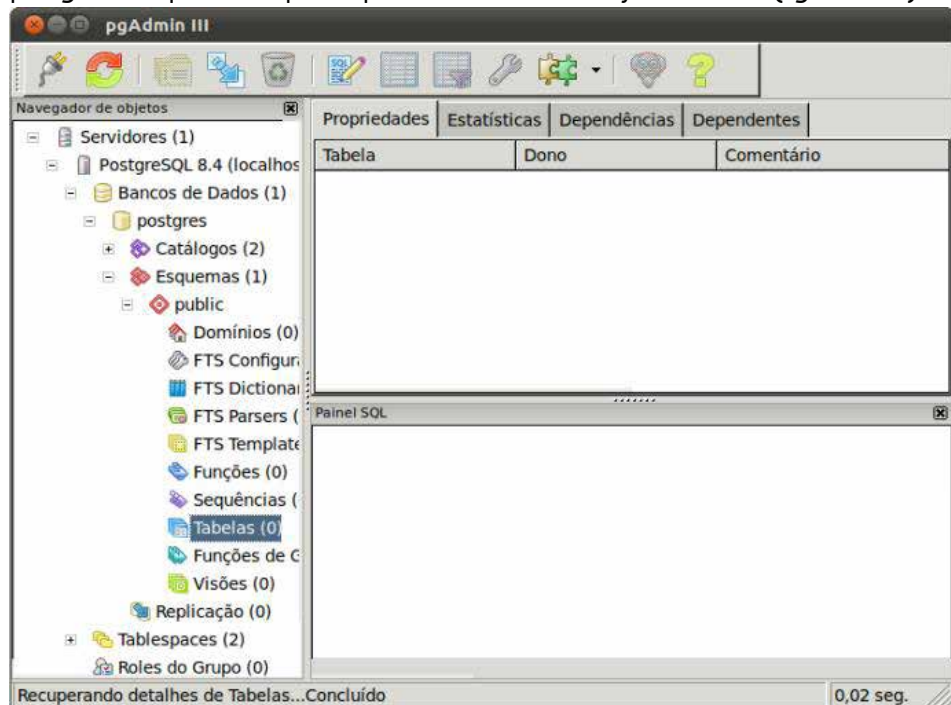


Figura 3.17 – Acessando objeto Tabelas.

Tendo como referência o modelo lógico do capítulo anterior execute os passos apresentados na sequência para criar a tabela ESTOQUE. Estes mesmos passos devem ser seguidos para criar as tabelas FORMA_PGTO e CLIENTE.

1. Clicar com o botão inverso do mouse e escolher a opção **Nova Tabela...**
2. No campo **Nome** da guia **Propriedades** digitar ESTOQUE.
3. Na guia **Colunas** clicar no botão **Adicionar**.
4. Na janela **Nova Coluna** preencher como mostrado na figura 3.18 e clicar no botão **OK**.

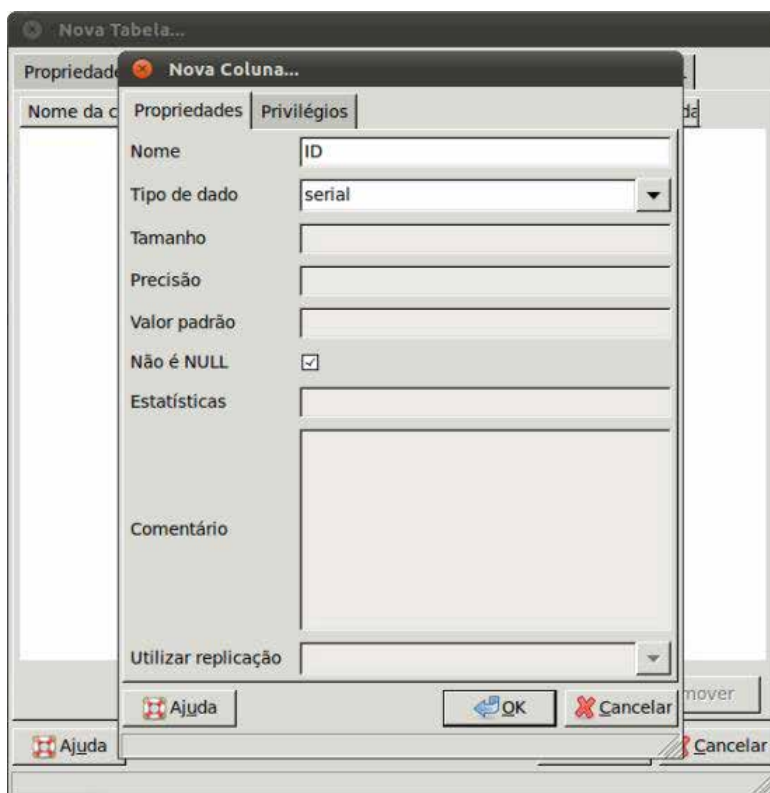


Figura 3.18 – Coluna ID da tabela ESTOQUE.

5. Clicar novamente no botão **Adicionar** e preencher como mostrado na figura 3.19.

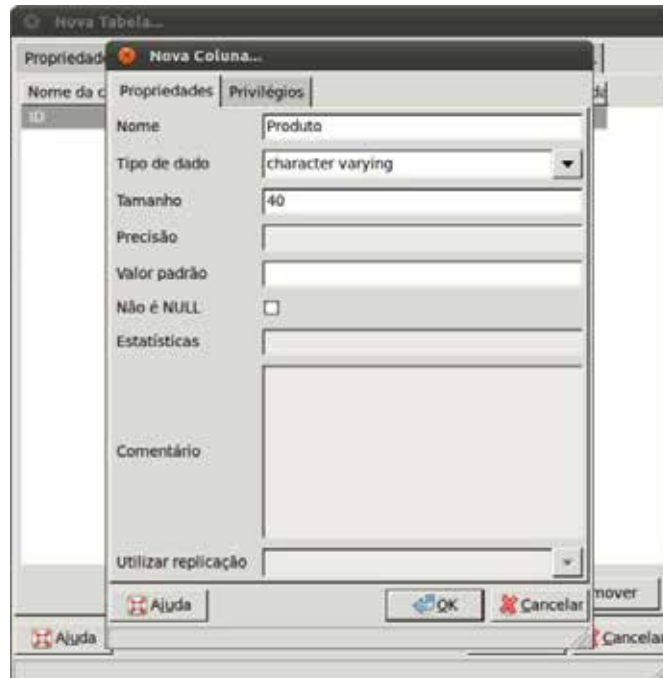


Figura 3.19 – Coluna Produto da tabela ESTOQUE.

6. Clicar outra vez no botão **Adicionar** e preencher como mostrado na figura 3.20.

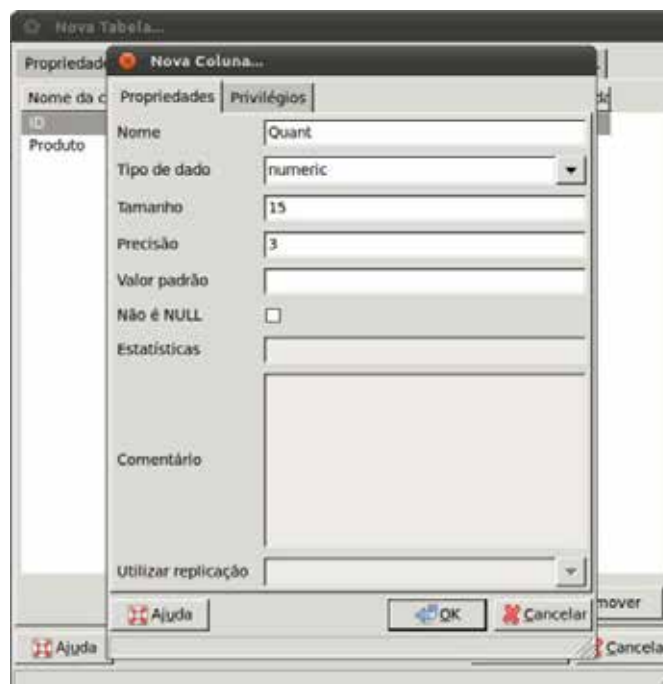


Figura 3.20 – Coluna Quant da tabela ESTOQUE.

7. A criação dos campos Vr_Compra e Vr_Venda é semelhante ao campo Quant criado no passo 6.
8. Na guia **Restrições** com a opção Chave Primária selecionada clicar no botão **Adicionar** (figura 3.21).
9. Na guia **Colunas** da janela **Nova Chave Primária...** selecionar o campo ID, clicar no botão **Adicionar** e em seguida no botão **OK**.
10. Na janela **Nova Tabela...** clicar no botão **OK**.

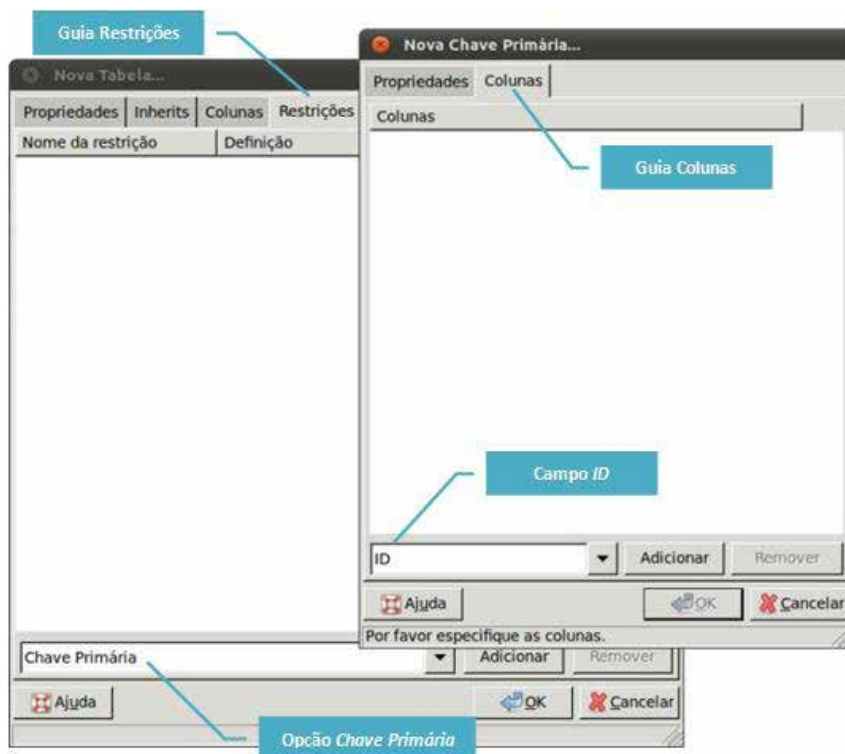


Figura 3.21 – Especificando a chave primária.

As figuras 3.18 e 3.19 trazem dois tipos diferentes dos utilizados no modelo lógico. Os tipos usados no modelo lógico são aceitos por qualquer SGBDR. Se na criação da coluna for utilizado o tipo Varchar, o PostgreSQL converterá o mesmo para seu tipo nativo Character Varying.

Da mesma maneira o tipo Serial é um tipo nativo do PostgreSQL, que neste caso é equivalente a um tipo inteiro com uma sequência criada de maneira implícita. Desta maneira o campo ID se torna um campo auto-incremente.

2.1.2. Criando a Tabela PDV

Os passos para criar a tabela PDV, suas colunas e a chave primária são similares aos da tabela ESTOQUE. A diferença está no passo seguinte a criação

da chave primária, após criar a chave primária devem-se criar as chaves estrangeiras da tabela.

Três informações são essenciais para a criação de uma chave estrangeira, (i) o campo da tabela local que será a chave estrangeira, (ii) a tabela estrangeira e (iii) o campo da tabela estrangeira que será referenciado. Essas informações podem ser passadas de maneira visual no pgAdmin.

Após criar todos os campos apresentados no modelo lógico e a chave primária, execute os passos apresentados na sequência. Estes mesmos passos devem ser executados para as tabelas ITEM_PDV e TELEFONES.

Na guia Restrições com a opção Chave Estrangeira selecionada clicar no botão Adicionar (figura 3.22).

Escolher a opção FORMA_PGTO (tabela estrangeira) no campo Referências da guia Propriedades da janela Nova Chave Estrangeira...

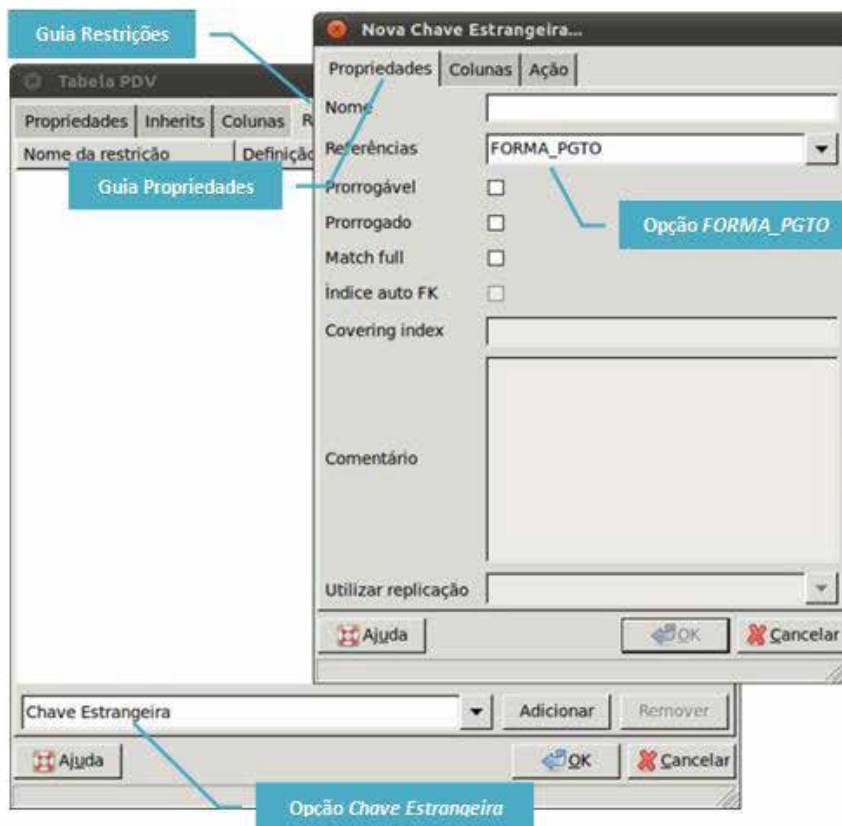


Figura 3.22 – Tabela referência para a chave estrangeira.

3. Na guia Colunas, nos campos Coluna local e Referenciando informar respectivamente ID_Forma_Pgto (campo da tabela local que será a chave estrangeira) e ID (campo da tabela estrangeira que será referenciado), clicar no botão adicionar (figura 3.23).



Figura 3.23 – Coluna local e referenciada para a chave estrangeira.

4. Na guia Ação, marcar as opções de acordo com a figura 3.24 e clicar no botão OK.

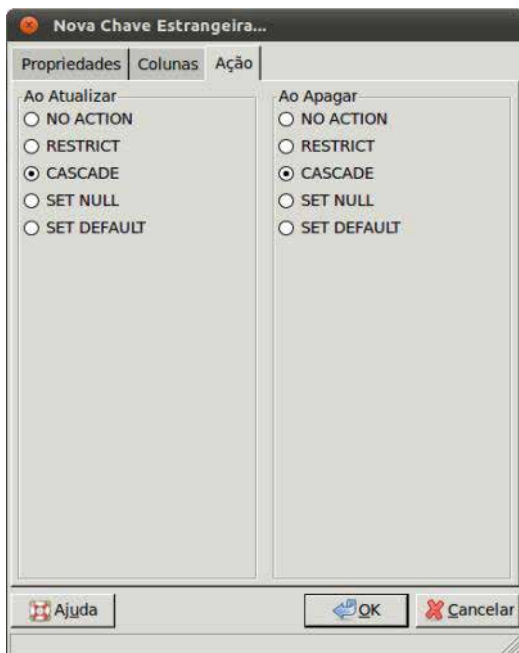


Figura 3.24 – Ações para a chave estrangeira.

As opções CASCADE do campo Ao Atualizar (figura 3.23) informa ao SGBD que caso um ID da tabela estrangeira FORMA_PGTO seja atualizado, todos os campos ID_Forma_Pgto da tabela local PDV que tenham o mesmo valor serão atualizados.

Já a opção CASCADE do campo Ao Apagar informa ao SGBD que caso um registro da tabela estrangeira FORMA_PGTO seja excluído, todos os registros da tabela local PDV com o valor do campo ID_Forma_Pgto igual ao ID do registro excluído também serão excluídos.

Repetir os passo da criação da chave estrangeira do campo ID_Forma_Pgto para criar a chave estrangeira do campo ID_Cliente. No final clicar no botão **OK** da janela **Nova Tabela...**

4

Capítulo

Introdução a SQL

Objetivo

- Com a evolução dos bancos de dados, surgiu uma linguagem descritiva que com o tempo foi se tornando um padrão para os bancos de dados relacionais. A SQL é uma linguagem declarativa criada para trabalhar exclusivamente com banco de dados, o seu conhecimento se torna indispensável para a atual conjuntura dos bancos de dados. Este capítulo é dedicado a história da SQL e aos comandos do grupo DDL e DML, além do comando SELECT e sua cláusula WHERE.

Introdução

Este é o primeiro de dois capítulos dedicados a SQL (Structured Query Language – Linguagem Estruturada de Consulta), uma linguagem declarativa desenvolvida para os bancos de dados relacionais, que devido sua simplicidade e facilidade de uso tornou-se um padrão para banco de dados relacionais.

O grande diferencial da SQL em relação a outras linguagens de consulta está no seu paradigma, pois a SQL é uma linguagem declarativa e não uma linguagem procedural. Para os novos usuários ela parece ser um pouco complicada, mas isso ocorre devido ao desconhecimento do paradigma declarativo, assim, o aluno após absorver os princípios desse paradigma, observará que o ciclo de aprendizagem da SQL é bastante reduzido.

Uma boa justificativa para se ter dois capítulos dedicados a SQL é dado por [DEITEL, 2010] quando afirma que “os sistemas de banco de dados atuais mais populares são os banco de dados relacionais” e ele conclui informando que a SQL “é a linguagem padrão internacional utilizada quase universalmente com banco de dados relacionais para realizar consultas e manipular dados”.

Neste capítulo as sintaxes dos comandos e sub-comandos da SQL serão apresentados baseados no padrão EBNF (Extended Backus-Naur Form – Forma Backus-Naur Estendida).

1. História

A SQL permitiu padronizar a construção e acesso a SGBDR (Sistema de Gerenciamento de Bancos de Dados Relacional) de diferentes tipos e em diferentes plataformas de software e hardware.

Essa padronização impulsionou não apenas a disseminação dos SGBDR, mas também a própria SQL. Para entender a importância da SQL são mostrados nesta subseção todos os pontos que levaram os DBAs (Da-

A BNF foi criada por JOHN BACKUS para descrever o ALGOL 58 e modificada ligeiramente por PETER NAUR para descrever o ALGOL 60. Algumas inconveniências causaram a extensão da BNF dando origem a EBNF.

O prêmio ACM TURING AWARD é conferido a uma pessoa que tenha dado contribuições de natureza técnica a comunidade computacional.

tabase Administrator – Administrador de Banco de Dados) a ter na SQL um aliado importante.

No final da década de 1960 o matemático Edgar Frank Codd apresentou as primeiras idéias sobre banco de dados relacional. Em junho de 1970 publicou o artigo “A relational model of data for large shared data banks – Um modelo relacional de dados para grandes bancos de dados compartilhados”, o que lhe rendeu em 1981 o prêmio ACM TURING AWARD.

Em 1973 a IBM criou o seu primeiro gerenciador de dados relacional, o SYSTEM R que utilizava a linguagem de consulta SEQUEL (Structured English Query Language – Linguagem Inglesa Estruturada de Consulta). Por motivos legais, a sigla foi alterada para SQL, mas o primeiro SGBDR disponível comercialmente foi o ORACLE em 1979.

A primeira versão padronizada da SQL ocorreu em 1986, ficando conhecida como SQL-86. Esse padrão foi inicialmente desenvolvido no âmbito da ANSI (American National Standards Institute – Instituto Nacional Americano de Padrões) sendo aprovado pela ISO (International Organization for Standardization – Organização Internacional para Padronização) em 1987.

Em 1989 foi publicada uma extensão do padrão SQL-86 chamada de SQL-89. A SQL-92, também chamada de SQL2, foi publicado em 1992 e aprovado pela ISO. Essa versão da SQL foi dividida em três partes:

1. Entry Level (Nível de Entrada) – Nesse nível foi definido um conjunto mínimo de comando para ser considerado padrão SQL;
2. Intermediate level (Nível Intermediário);
3. Full (Completo).

A SQL-99 ou SQL3 foi aprovada pela ISO no final do ano de 1999. Nela foram definidos os usos de triggers, stored procedures, consultas recursivas, entre outros. Esse padrão também definiu regras para os SGBDOR (Sistema de Gerenciamento de Bancos de Dados Objeto-Relacional) (Ler 1.2. Modelos de Bancos de Dados), implementando assim o suporte ao tratamento de objetos.

No ano de 2003 foi lançado o SQL-2003, introduzindo características relacionadas ao XML (eXtensible Markup Language – Linguagem de Marcação Extensiva), sequências padronizadas e colunas com valores de auto-generalização.

A versão SQL-2008 trouxe nas especificações formas para a SQL poder ser usada em conjunto com XML, incluindo importação, armazenamento, manipulação e publicação de dados XML no SGBDR.

A SQL é uma linguagem padronizada, mas cada SGBDR apresenta dialeto próprio, com extensões diferentes entre cada fabricante de SGBD.

XML é uma linguagem universal usada para troca de informações entre organizações, empresas, departamentos e banco de dados – entre outros – de uma forma transparente e organizada, permitindo ao desenvolvedor criar as marcações (tags) mais adequadas para cada situação.

2. Grupos

Os comandos da SQL são tradicionalmente separados em dois grupos:

- **DDL (Data Definition Language – Linguagem de Definição de Dados):** Subconjunto utilizado para criar, alterar e excluir tabelas e elementos associados: esse é o grupo que mais muda de um fabricante para outro.
- **DML (Data Manipulation Language – Linguagem de Manipulação de Dados):** Subconjunto dos comandos usado para inserir, atualizar e apagar dados.

Para recuperar (consultar) os dados utiliza-se o comando select. Alguns autores incluem esse comando dentro do grupo DML, uma vez que para recuperar os dados é necessário manipulá-los, sem necessariamente ter que alterar seu estado. Outros já preferem definir um grupo específico para ele chamado DQL (Data Query Language – Linguagem de Consulta de Dados).



2.1. Outros Grupos

Além da divisão tradicional, é possível ver outras divisões que foram criadas no decorrer do tempo:

- **DCL (Data Control Language – Linguagem de Controle de Dados):** Subconjunto de comandos que controla o acesso dos usuários aos dados.
- **DTL (Data Transaction Language - Linguagem de Transação de Dados):** Subconjuntos de comandos usados para iniciar e finalizar transações.
- **DQL (Data Query Language – Linguagem de Consulta de Dados):** Com apenas um único comando – select – e suas várias cláusulas e opções – nem sempre obrigatórias – permite recuperar os dados de uma ou mais tabelas através de consultas elaboradas como uma descrição do resultado desejado.

Além desses grupos de comandos a SQL tem operadores lógicos, operadores relacionais e funções de agregação que, assim como na DDL, podem mudar de um fabricante para outro.

3. SQL Editor

Para executar os exemplos apresentados neste capítulo será usado o SQL Editor do pgAdmin. Com o banco de dados desejado selecionado no navegador de objetos, deve-se clicar no botão  para abrir o SQL Editor (figura 4.1). Após escrever os exemplos apresentados neste capítulo, deve-se clicar no botão  para executar o comando digitado.

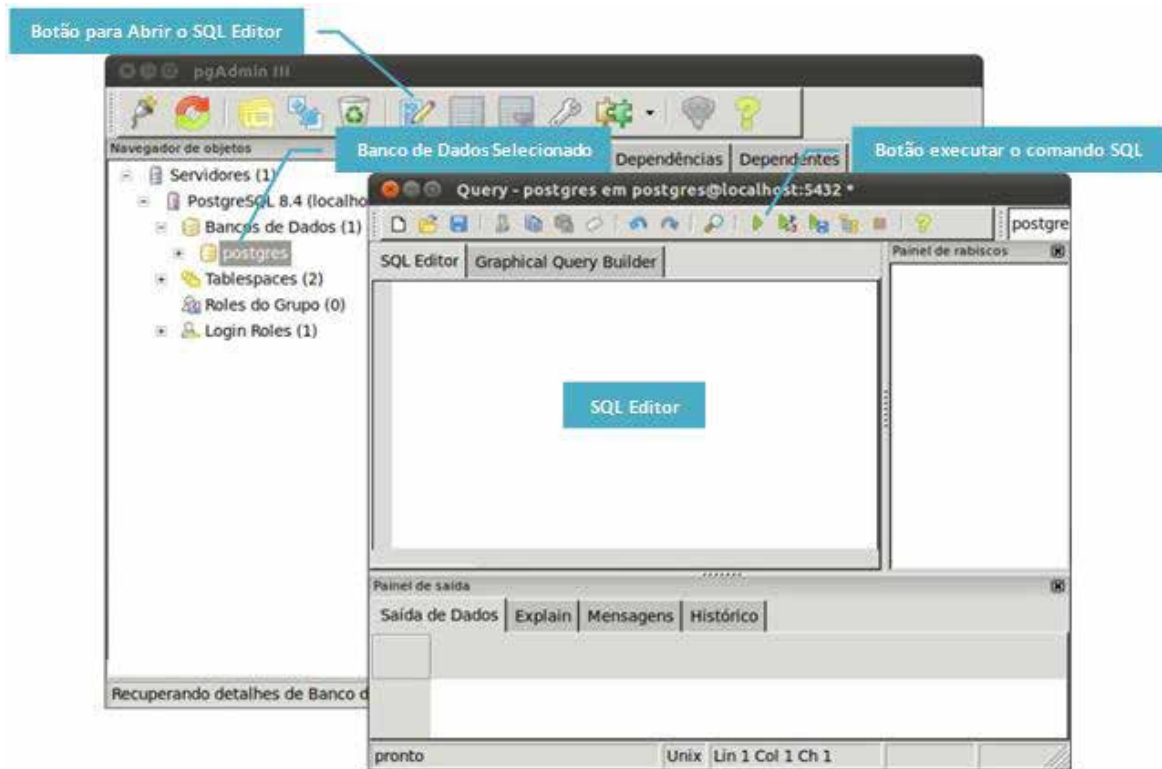


Figura 4.1 – Editor de SQL.

A SQL é *CASE INSENSITIVE* (não diferencia letras maiúsculas de minúsculas). Mas vale ressaltar que os dados armazenados podem ser *CASE SENSITIVE* (diferencia letras maiúsculas de minúsculas), este fato depende do SGBDR.

Na EBNF os *COLCHETES* denotam uma parte opcional e as *CHAVES* indicam que partes podem ser repetidas indefinidamente ou omitidas completamente.

Os exemplos apresentados neste capítulo não fazem parte do projeto apresentado nos capítulos 2 e 3. Serão apresentados exemplos didáticos para melhor entendimento da SQL, independente do projeto onde serão usados.

4. DDL – Parte 1

A DDL é o grupo que mais muda de um dialeto para o outro, mas vale ressaltar que a sintaxe apresentada aqui é muito semelhante a maioria dos SGBDRs (Sistemas Gerenciadores de Bancos de Dados Relacionais).

4.1. Comando CREATE TABLE

O comando CREATE TABLE é usado para criar uma tabela onde os dados serão armazenados.

Sintaxe:

```
create table Tabela (
    Atributo Tipo_Dado [Restrição]
```

```
[ {, Atributo Tipo_Dado [Restrição] } ]
);
```

Rótulo	Descrição
Tabela	Nome da tabela
Atributo	Nome do atributo
Tipo_Dado	Tipo de dados do atributo
Restrição	Restrição de dados para o atributo

TIPOS DE DADOS MAIS COMUNS AOS SGBDRS	
Char(n)	Caractere de tamanho fixo
Varchar(n)	Caractere de tamanho variável
Number(n, p)	Números de ponto flutuante com total de dígitos n e total de dígitos à direita do ponto decimal p
Integer	Números inteiros – Alguns SGBDR utilizam Number(n) para identificar um número inteiro
Date	Armazena data – Alguns SGBDR armazenam data e hora

Exemplo₁:

```
create table Setor (
    ID integer not null primary key,
    Setor varchar(20)
);
```

Neste exemplo foi criada a tabela Setor, esta tabela tem os atributos (i) ID do tipo inteiro e (ii) Setor que pode armazenar até 20 caracteres. Um detalhe interessante neste exemplo é o atributo Setor que é homônimo da tabela, isso é possível porque a sintaxe da SQL é bem elaborada, deixando claro quem é a tabela e quem é atributo.

A restrição NOT NULL colocada no atributo ID informa ao SGBDR que só pode aceitar um registro nesta tabela quando esse atributo for informado, ou seja, o atributo ID não pode ser nulo. Também foi informado ao SGBDR que o atributo ID é uma chave primária através do sub-comando PRIMARY KEY.

A figura 4.2 mostra como o **Editor de SQL** fica após o comando anterior ser executado com êxito.

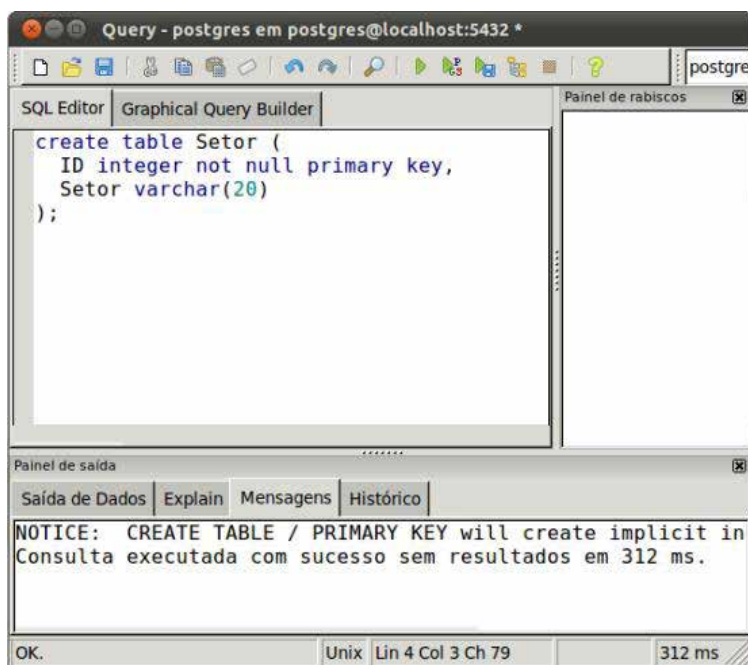


Figura 4.2 – Comando CREATE após executado no Editor de SQL.

Exemplo₂:

```
create table Funcion (
    ID integer not null primary key,
    Nome varchar(40),
    ID_Setor integer,
    primary key(ID),
    foreign key(ID_Setor) references Setor (ID) on delete cascade
);
```

Esse comando cria a tabela Funcion, com os atributos ID, Nome e ID_Setor. Mas neste exemplo a chave primária foi criada de outra forma, o sub-comando PRIMARY KEY foi inserido após o último atributo (ID_Setor) e foi colocado entre parênteses o campo que representa a chave primária.

Se a chave primária for simples (composta por apenas um único atributo), pode-se usar dos dois modos apresentados nos exemplos. Mas se a chave primária for composta (composta por dois ou mais atributos), deve-se usar o modo apresentado no segundo exemplo, separando os campos com vírgula como mostrado na sintaxe abaixo:

```
create table Tabela (
    Atributo Tipo not null,
    Atributo Tipo not null [ {, Atributo Tipo [Restrição] } ] ,
    primary key( Atributo [ {, Atributo} ] )
);
```

Neste exemplo também foi criada a chave estrangeira através do sub-comando FOREIGN KEY. Para criar uma chave estrangeira junto com o comando CREATE TABLE, a tabela estrangeira já deve estar criada.

A semântica para criar a chave estrangeira é a seguinte:

- Após o sub-comando FOREIGN KEY deve-se colocar entre parênteses o atributo da tabela que será a chave estrangeira;
- Após a palavra reservada REFERENCES deve-se informar a tabela que está sendo referenciada pela chave estrangeira, e entre parênteses o atributo da tabela referenciada;
- As palavras reservadas ON DELETE CASCADE indicam ao SGBDR que após excluir um registro da tabela referenciada, todos os registros da tabela atual que tem chave estrangeira com o mesmo valor serão excluídos (Leia 3.2.1.2. Criando a Tabela PDV).

4.2. Comando ALTER TABLE

O comando ALTER TABLE é usado para fazer alterações em tabelas já existentes no banco de dados.

4.2.1. Adicionar Atributo a Tabela

Para adicionar um ou mais atributos numa tabela já existente, deve-se usar o comando ALTER TABLE em conjunto com o sub-comando ADD.

Sintaxe:

```
alter table Tabela  
add Atributo Tipo_Dado [Restrição]  
[ {, add Atributo Tipo_Dado [Restrição] } ];
```

Exemplo:

```
alter table Funcion  
add Salario numeric(7, 2),  
add Dt_Nasc date;
```

Este exemplo adiciona os atributos (i) Salario do tipo numérico de tamanho 7 com precisão 2 e (ii) Dt_Nasc do tipo data.

4.2.2. Modificar Tipo de Dados do Atributo

Para modificar o tipo de dados de um ou mais atributos já existentes numa tabela, deve-se usar o comando ALTER TABLE em conjunto com os sub-comandos ALTER e TYPE.

Sintaxe:

```
alter table Tabela  
alter Atributo type Tipo_Dado  
[ {, alter Atributo type Tipo_Dado } ];
```

Exemplo:

```
alter table Funcion  
alter Dt_Nasc type varchar(10);
```

A mudança do tipo de dados do atributo Dt_Nasc de date para varchar(10) só foi possível devido eles serem compatíveis. Se fosse tentando

modificar o tipo date para varchar(5) seria gerado um erro, pois o tipo date não pode ser convertido no tipo varchar(5) sem que haja a perda de dados armazenados.

4.2.3. Excluir Atributo da Tabela

Para excluir um ou mais atributos numa tabela já existente, deve-se usar o comando ALTER TABLE em conjunto com o sub-comando DROP.

Sintaxe:

```
alter table Tabela  
drop Atributo  
[ {, drop Atributo } ];
```

Exemplo:

```
alter table Funcion  
drop Dt_Nasc;
```

Os sub-comando DROP está informando ao SGBD que o atributo Dt_Nasc da tabela Funcion deve ser excluído.

4.2.4. Adicionar Chave Primária a Tabela

Para incluir uma chave primária numa tabela já existente, deve-se usar o comando ALTER TABLE em conjunto com o sub-comando PRIMARY KEY.

Sintaxe:

```
alter table Tabela add primary key( Atributo [, Atributo] ) ;
```

Exemplo:

```
alter table Funcion add primary key(ID);
```

Neste exemplo é adicionada a chave primária na tabela Funcion. Mas o exemplo só será executado com êxito se a tabela mencionada ainda não tiver chave primária declarada.

4.2.5. Adicionar Chave Estrangeira a Tabela

Para incluir uma ou mais chaves estrangeiras numa tabela já existente, deve-se usar o comando ALTER TABLE em conjunto com o sub-comando FOREIGN KEY.

Sintaxe:

```
alter table Tabela foreign key(Atributo) references Tabela_FK (Atributo_FK);
```

Exemplo:

```
alter table Funcion foreign key(ID_Setor) references Setor (ID);
```

Aqui é adicionada uma chave estrangeira na tabela Funcion. A vantagem dessa abordagem sobre a apresentada anteriormente é a flexibilidade de poder criar todas as tabelas, para só depois criar os relacionamentos existente entre elas, independente da ordem que as tabelas foram criadas.

4.3. Comando DROP TABLE

O comando DROP TABLE é usado para excluir tabelas do banco de dados.

Sintaxe:

```
drop table Tabela;
```

Exemplo:

```
drop table Funcion;
```

A tabela Funcion e todos os seus registros são excluídos do banco de dados com este exemplo.

4.5. DML

As inclusões, alterações e exclusões de dados numa tabela são feitas pelos comandos do grupo DML. As sintaxes apresentadas nesta seção pertencem ao SQL padrão, mas assim como os comandos DDL, alguns SGBDRs podem apresentar dialetos diferentes.

4.5.1. Preparando o Banco de Dados

Antes de mostrar os comandos DML é preciso excluir todas as tabelas do banco de dados através do comando DROP TABLE, para evitar conflitos de tabelas.

Após excluir todas as tabelas do banco de dados devem-se criar as tabelas que serão usadas nos exemplos. Na sequência são apresentadas as tabelas:

- **Setor:** Armazenará os dados dos setores onde os funcionários da empresa podem ser lotados. Esta tabela tem o atributo ID que é a chave primária e o atributo Setor (homônimo da tabela) guardado o nome do setor.
- **Funcion:** Armazenará os dados dos funcionários da empresa. O atributo ID é a chave primária, o nome do funcionário deve ser informado no atributo Nome. O salário mensal do funcionário é gravado no atributo Salario. O relacionamento entre o funcionário e o setor onde está lotado é representado pela chave estrangeira ID_Setor.
- **Cliente:** Armazenará os dados dos clientes da empresa. A chave primária do cliente é representada pelo atributo ID. O nome e o sobrenome do cliente ficam guardados nos atributo Nome e Sobrenome, respectivamente.
- **Pedido:** Armazenará os dados dos pedidos. O atributo ID registrará a chave primária do pedido. O valor total do pedido será guardado no atributo Valor. O atributo ID_Cliente é a chave estrangeira responsável por fazer o relacionamento entre o pedido e o cliente.

A figura 4.3 mostra o modelo lógico utilizado nos exemplos que se seguem.

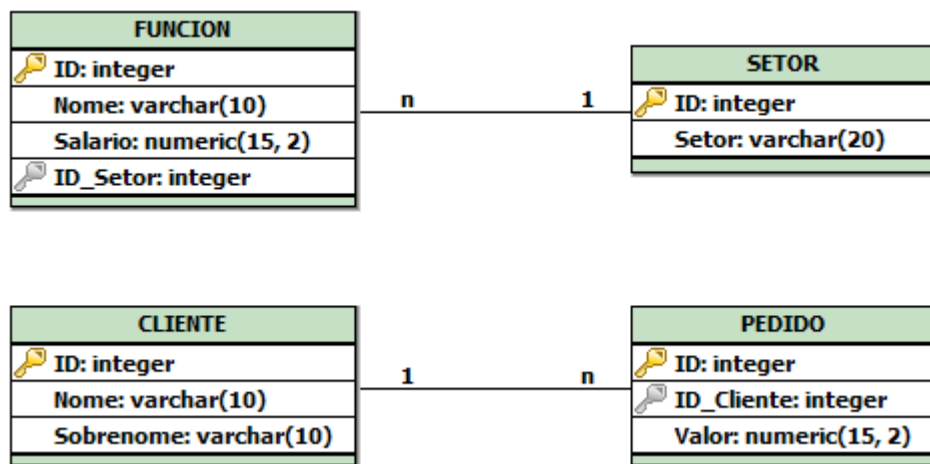


Figura 4.3 – Modelo lógico.

O modelo físico baseado no modelo lógico apresentado anteriormente é mostrado na figura 4.4.

```
create table Funcao (
  ID integer not null primary key,
  Nome varchar(10),
  Salario numeric(15, 2),
  ID_Setor integer
);

create table Setor (
  ID integer not null primary key,
  Setor varchar(20)
);

create table cliente (
  ID integer not null primary key,
  Nome varchar(10),
  Sobrenome varchar(10)
);

create table Pedido (
  ID integer not null primary key,
  ID_Cliente integer,
  valor numeric(15, 2),
  foreign key(ID_Cliente) references cliente(ID)
);

alter table Funcao add foreign key(ID_Setor) references Setor(ID);
```

Figura 4.4 – Modelo físico.

4.5.2. Comando INSERT

O comando INSERT adiciona um novo registro numa tabela.

Sintaxe:

```
insert into Tabela [ (Atributo [ {, Atributo} ] ) ]
values (Valor [ {, Valor} ] );
```

Rótulo	Descrição
Tabela	Nome da tabela
Atributo	Nome do atributo
Valor	Valor que será inserido no atributo

As tabelas criadas usando o modelo físico da figura 4.4, serão povoadas com os comandos INSERTs dos exemplos mostrados na continuação, e ficarão como mostradas na figura 4.5.

ID	Nome	Salario	ID_Setor
1	Tadeu	1500,00	1
2	Ylane	1200,00	2
3	Julian	1000,00	1
4	Ewerton	1000,00	1
5	João	800,00	2
6	Celestino	1500,00	3
7	Maria	500,00	null
8	Joana	1000,00	4
9	Fernanda	1000,00	4

ID	Setor
1	Desenvolvedor
2	Manutenção
3	Financeiro

ID	Nome	Sobrenome
1	Francisco	Silva
2	José	Lima
3	Maria	Silva
4	Adriana	Ferreira
5	João	Oliveira
6	Eduardo	Souza

ID	ID_Cliente	Valor
1	2	1000,00
2	4	2000,00
3	2	1500,00
4	5	2500,00
5	2	1000,00

Figura 4.5 – Tabelas povoadas.

Exemplos₁ – Inserção com atributos explícitos:

```
insert into Setor (ID, Setor)
values (1, 'Desenvolvedor');
```

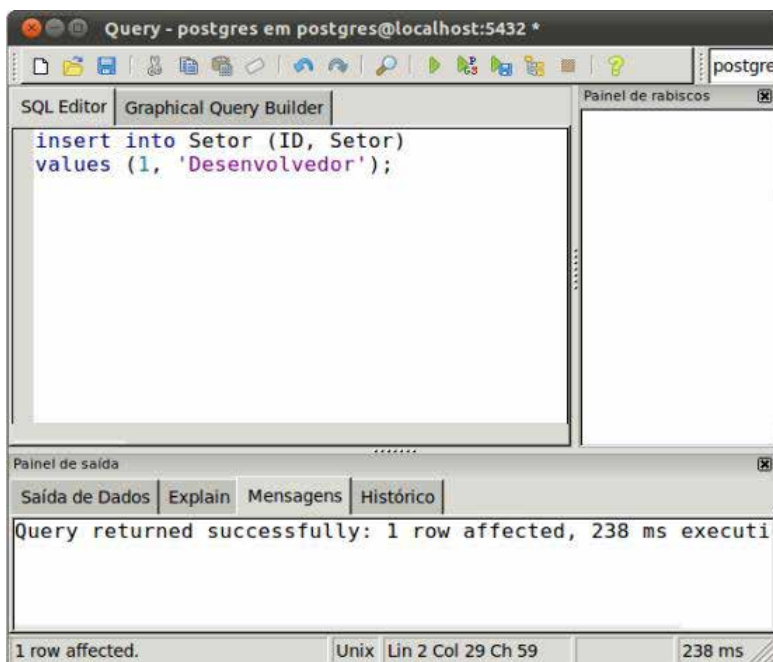


Figura 4.6 – Comando INSERT após executado no Editor de SQL.

Este exemplo é incluído na tabela Setor um registro onde o atributo ID receber o valor 1, e o atributo Setor receber o valor Desenvolvedor.

Os valores dos atributos que armazenam caracteres devem ser colocados entre apóstrofos, assim como os valores dos atributos que armazenam datas. Já os atributos que armazenam números podem ser informados sem os apóstrofos.

A figura 4.6 mostra como o **Editor de SQL** fica após o comando anterior ser executado com êxito.

Exemplos₂ – Inserção com atributos implícitos:

```
insert into Setor  
values (2, 'Manutenção');
```

Neste caso o primeiro atributo da tabela (ID) recebe o valor 2 e o segundo atributo da tabela (Setor) recebe o valor Manutenção.

Embora a SQL não faça diferença entre estes dois modos de inserção, as boas práticas recomenda utilizar o modo explícito motivos como os apresentados na sequência:

- A inclusão de novos atributos na tabela, de uma forma geral, não obriga a alteração do comando;
- A alteração na posição de atributos da tabela não obriga a alteração do comando;
- Permite a inclusão de registros com atributos nulos, desde que o atributo não tenha a restrição NOT NULL;
- O comando é executado de imediato, uma vez que o SGBDR não vai precisar pesquisar quais são os atributos da tabela utilizada, antes de executar o comando INSERT.

A figura 4.7 apresenta os outros comandos INSERTs que devem ser usados para povoar a tabela Setor.

```
insert into Setor (ID, Setor)  
values (3, 'Financeiro');  
  
insert into Setor (ID, Setor)  
values (4, 'vendas');
```

Figura 4.7 – Povoando a tabela Setor.

A figura 4.8 apresenta os comandos INSERTs usados para povoar a tabela Funcion. O detalhe desta figura fica por conta da inclusão do sétimo registro; nesta inclusão o atributo ID_Setor não foi informado, com isto este registro terá o valor NULO conferido ao atributo ID_Setor.

Ainda sobre a figura 4.8, só é possível informar no atributo ID_Setor valores que tenham referências na tabela Setor, uma vez que o atributo ID_Setor é uma chave estrangeira. Caso seja informado no ID_Setor um valor não existente na tabela Setor o SGBDR irá gerar um erro de integridade.

```
insert into Funcion (ID, Nome, Salario, ID_Setor)
values (1, 'Tadeu', 1500, 1);

insert into Funcion (ID, Nome, Salario, ID_Setor)
values (2, 'Ylane', 1200, 2);

insert into Funcion (ID, Nome, Salario, ID_Setor)
values (3, 'Julian', 1000, 1);

insert into Funcion (ID, Nome, Salario, ID_Setor)
values (4, 'Ewerton', 1000, 1);

insert into Funcion (ID, Nome, Salario, ID_Setor)
values (5, 'João', 800, 2);

insert into Funcion (ID, Nome, Salario, ID_Setor)
values (6, 'Celestino', 1500, 3);

insert into Funcion (ID, Nome, Salario)
values (7, 'Maria', 500);

insert into Funcion (ID, Nome, Salario, ID_Setor)
values (8, 'Joana', 1000, 4);

insert into Funcion
values (9, 'Fernanda', 1000, 4);
```

Figura 4.8 – Povoando a tabela Setor.

A figura 4.9 apresenta os comandos INSERTs usados para povoar a tabela Cliente.

```
insert into Cliente (ID, Nome, Sobrenome)
values (1, 'Francisco', 'Silva');

insert into Cliente (ID, Nome, Sobrenome)
values (2, 'José', 'Lima');

insert into Cliente (ID, Nome, Sobrenome)
values (3, 'Maria', 'Silva');

insert into Cliente (ID, Nome, Sobrenome)
values (4, 'Adriana', 'Ferreira');

insert into Cliente (ID, Nome, Sobrenome)
values (5, 'João', 'Oliveira');

insert into Cliente (ID, Nome, Sobrenome)
values (6, 'Eduarda', 'Souza');
```

Figura 4.9 – Povoando a tabela Cliente.

A figura 4.10 apresenta os comandos INSERTs usados para povoar a tabela Pedido. O atributo ID_Cliente da tabela Pedido é uma chave estrangei-

ra que faz referência a tabela Cliente, logo só é possível informar para este atributo valores devidamente referenciados na tabela Cliente.

```
insert into Pedido (ID, ID_Cliente, valor)
values (1, 2, 1000);

insert into Pedido (ID, ID_Cliente, valor)
values (2, 4, 2000);

insert into Pedido (ID, ID_Cliente, valor)
values (3, 2, 1500);

insert into Pedido (ID, ID_Cliente, valor)
values (4, 5, 2500);

insert into Pedido (ID, ID_Cliente, valor)
values (5, 2, 1000);
```

A cláusula *WHERE* será mais bem explicada na seção 4.6.

Figura 4.10 – Povoando a tabela Pedido.

4.5.3. Comando UPDATE

O comando UPDATE altera um ou mais registros. Os registros que serão alterados dependem do filtro incluído na cláusula WHERE.

Sintaxe:

```
update Tabela set Atributo = Valor [ {, Atributo = Valor} ]
[ where Condição ];
```

Exemplo₁:

```
update Funcion set Salario = 1200
where ID_Setor = 4;
```

Com este comando todos os registros da tabela Funcion com o atributo ID_Setor igual a 4 terão o atributo Salario alterado para 1.200,00. Se a cláusula WHERE for omitida, todos os registros da tabela Funcion terão o atributo Salario alterado para 1.200,00.

Exemplo₂:

```
update Funcion set Salario = Salario * 1.5;
where ID_Setor = 4;
```

Com este comando todos os registros da tabela Funcion com o atributo ID_Setor igual a 4 terão o atributo Salario aumentado em 50%, ou seja, altera o valor do atributo de 1.200,00 para 1.800,00.

4.5.4. Comando DELETE

O comando DELETE exclui um ou mais registros. Os registros que serão excluídos dependem do filtro incluído na cláusula WHERE.

Sintaxe:

```
delete from Tabela  
[ where Condição ];
```

Exemplo:

```
delete from Funcion  
ID_Setor = 4;
```

Este comando exclui todos os registros da tabela Funcion com o atributo ID_Setor igual a 4. Se a cláusula WHERE for omitida, todos os registros da tabela Funcion serão excluídos.

4.5.5. Comando COMMIT

Após executar um ou mais dos comandos de manipulação de dados apresentados (INSERT, UPDATE, DELETE) os dados ficam na memória cache da transação, para os dados serem persistidos no banco de dados é necessário utilizar o comando COMMIT.

O pgAdmin efetua o COMMIT automaticamente após a execução dos comandos, mas isso é uma característica deste front-end, o mesmo pode não acontecer com outros front-ends.

Sintaxe:

```
commit;
```

Exemplo:

```
commit;
```

4.5.6. Comando ROLLBACK

Para descartar os dados que estão na memória cache da transação, deve-se usar o comando ROLLBACK.

Sintaxe:

```
rollback;
```

Exemplo:

```
rollback;
```

4.6. Comando SELECT – Parte 1

A recuperação dos dados armazenados no banco de dados é efetuada através do comando SELECT. Este comando pode recuperar os dados de uma ou mais tabelas, sendo um dos comandos mais simples e, ao mesmo tempo, mais extenso da SQL devido as suas funções, operandos, comandos, sub-comandos e cláusulas não obrigatórias.

4.6.1. Comando SELECT simples

Este comando recupera todos os registros de uma tabela.

Sintaxe:

```
select Atributo [ {, Atributo} ] from Tabela;
```

Exemplo1:

```
select ID, Nome from Funcion;
```

O resultado deste exemplo trará os atributos ID e Nome de todos os registros da tabela Funcion.

Resultado₁:

ID	Nome
1	Tadeu
2	Ylane
3	Julian
4	Ewerton
5	João
6	Celestino
7	Maria

A figura 4.11 mostra como o Editor de SQL fica após o comando anterior ser executado com êxito.

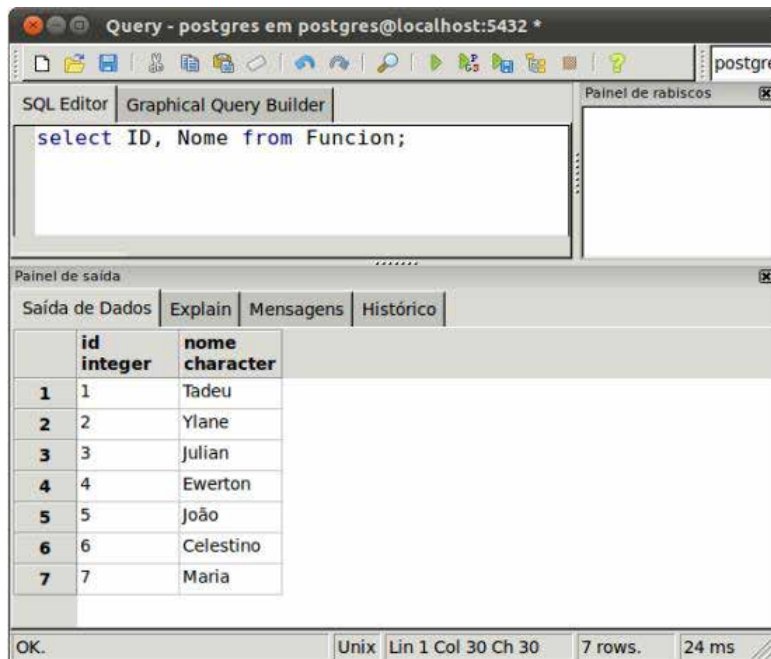


Figura 4.11 – Comando SELECT após executado no Editor de SQL.

Exemplo₂:

```
select * from Funcion;
```

O resultado deste exemplo trará todos os atributos de todos os registros da tabela Funcion.

Resultado₂:

ID	Nome	Salario	ID_Setor
1	Tadeu	1500,00	1
2	Ylane	1200,00	2
3	Julian	1000,00	1
4	Ewerton	1000,00	1
5	João	800,00	2
6	Celestino	1500,00	3
7	Maria	500,00	null

Neste exemplo foi inserido o caractere asterisco no lugar dos atributos. O asterisco é uma máscara que informa ao SGBDR que ele deve ser trocado por todos os atributos da tabela consultada.

4.6.2. Cláusula *WHERE* com Condições Simples

É possível filtrar registros para mostrar apenas os dados de interesse através da cláusula *WHERE* em conjunto com os operadores comparativos.

Sintaxe:

```
select Atributo [ {, Atributo} ] from Tabela  
[where Condição];
```

Embora a cláusula *WHERE* seja opcional, é altamente recomendável sempre usá-la para qualquer tipo de consulta que for ser realizada. Mas por questões didáticas, alguns exemplos apresentados neste capítulo iram omitir a cláusula *WHERE*.

4.6.2.1. Operador Comparativo “=” (Igual)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula *WHERE* igual ao valor informado após o operador “=”.

Exemplo₁:

```
select ID, Nome from Funcion  
where Nome = 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela *Funcion* onde o atributo *Nome* seja igual a “Tadeu”.

Resultado₁:

ID	Nome
1	Tadeu

Como comentado anteriormente, a SQL é CASE INSENSITIVE (não diferencia letras maiúsculas de minúsculas), logo o resultado será igual ao apresentado mesmo que o comando seja executado no exemplo2:

Exemplo₂:

```
SELECT ID, NOME FROM FUNCION  
WHERE NOME = 'Tadeu';
```

Já o resultado do comando mostrado no exemplo3 e exemplo4 depende do SGBDR, pois caso os dados armazenados sejam CASE SENSITIVE (diferencia letras maiúsculas de minúsculas) o seu resultado será vazio.

Exemplo₃:

```
select ID, Nome from Funcion  
where Nome = 'tadeu';
```

Exemplo₄:

```
select ID, Nome from Funcion  
where Nome = 'TADEU';
```

4.6.2.2. Operador Comparativo “<>” (Diferente)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE diferente do valor informado após o operador “<>”.

Exemplo:

```
select ID, Nome from Funcion  
where Nome <> 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja diferente de “Tadeu”.

Resultado:

ID	Nome
2	Ylane
3	Julian
4	Ewerton
5	João
6	Celestino
7	Maria

4.6.2.3. Operador Comparativo “>” (Maior que)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE maior que o valor informado após o operador “>”.

Exemplo:

```
select ID, Nome from Funcion  
where Nome > 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja maior que "Tadeu", ou seja, fiquem após "Tadeu" numa ordenação ascendente.

Resultado:

ID	Nome
2	Ylane

4.6.2.4. Operador Comparativo ">=" (Maior que ou Igual)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE maior ou igual ao valor informado após o operador ">=".

Exemplo:

```
select ID, Nome from Funcion
where Nome >= 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja maior ou igual a "Tadeu".

Resultado:

ID	Nome
1	Tadeu
2	Ylane

4.6.2.5. Operador Comparativo "<" (Menor que)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE menor que o valor informado após o operador "<".

Exemplo:

```
select ID, Nome from Funcion
where Nome < 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja menor que "Tadeu".

Resultado:

ID	Nome
3	Julian
4	Ewerton
5	João
6	Celestino
7	Maria

4.6.2.6. Operador Comparativo “<=” (Menor que ou Igual)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE menor ou igual ao valor informado após o operador “<=”.

Exemplo:

```
select ID, Nome from Funcion  
where Nome <= 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja menor ou igual a “Tadeu”.

Resultado:

ID	Nome
1	Tadeu
3	Julian
4	Ewerton
5	João
6	Celestino
7	Maria

4.6.2.7. Operador Comparativo “BETWEEN ... AND ...” (entre dois valores)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE (i) maior ou igual ao primeiro valor informado após o operador BETWEEN e (ii) menor ou igual ao segundo valor informado após o operador BETWEEN.

Exemplo:

```
select ID, Nome from Funcion  
where Nome between 'João' and 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja maior ou igual a “João” e menor ou igual a “Tadeu”.

Resultado:

ID	Nome
1	Tadeu
3	Julian
5	João
7	Maria

4.6.2.8. Operador Comparativo “NOT BETWEEN ... AND ...” (não está entre dois valores)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE (i) menor ou igual ao primeiro valor informado após o operador NOT BETWEEN e (ii) maior ou igual ao segundo valor informado após o operador NOT BETWEEN.

Exemplo:

```
select ID, Nome from Funcion  
where Nome not between 'João' and 'Tadeu';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja menor que “João” e maior que “Tadeu”.

Resultado:

ID	Nome
2	Ylane
4	Ewerton
6	Celestino

4.6.2.9. Operador Comparativo “in (lista)” (igual a qualquer valor da lista)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE igual a pelo menos um dos valores informados após o operador IN.

Exemplo:

```
select ID, Nome from Funcion  
where Nome in ('João','Tadeu');
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome igual a “João” ou igual a “Tadeu”.

Resultado:

ID	Nome
1	Tadeu
5	João

4.6.2.10. Operador Comparativo “not in (lista)” (diferente de qualquer valor da lista)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE diferente de todos os valores informados após o operador NOT IN.

Exemplo:

```
select ID, Nome from Funcion
where Nome not in ('João','Tadeu');
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja diferente de “João” e diferente de “Tadeu”.

Resultado:

ID	Nome
2	Ylane
3	Julian
4	Ewerton
6	Celestino
7	Maria

4.6.2.11. Operador Comparativo “like” (Igual a uma cadeia de caracteres)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE com uma cadeia de caracteres igual ao valor informado após o operador LIKE.

Exemplo₁:

```
select ID, Nome from Funcion
where Nome like 'J%';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome começa com o caractere “J”, independente do que venha após este caractere.

Resultado₁:

ID	Nome
3	Julian
5	João

Exemplo₂:

```
select ID, Nome from Funcion
where Nome like '%n';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome termina com o caractere “n”, independente do que venha antes deste caractere.

Resultado₂:

ID	Nome
3	Julian
4	Ewerton

Exemplo₃:

```
select ID, Nome from Funcion
where Nome like '%e%';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome tenha o caractere “e”, independente do que venha antes ou após este caractere.

Resultado₃:

ID	Nome
1	Tadeu
2	Ylane
4	Ewerton
6	Celestino

Exemplo₄:

```
select ID, Nome from Funcion  
where Nome like '%ria%';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome tenha a cadeia de caracteres “ria”, independente do que venha antes ou após esta cadeia de caracteres.

Resultado₄:

ID	Nome
7	Maria

A máscara usada nos exemplos do operador “like” foi o “%”, porém esta máscara pode mudar de um SGBDR para outro.

4.6.2.12. Operador Comparativo “not like”

(diferente de uma cadeia de caractere)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE com cadeias de caracteres diferentes do valor informado após o operador NOT LIKE.

Exemplo:

```
select ID, Nome from Funcion  
where Nome not like 'J%';
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome não começa com o caractere “J”, independente do que venha após este caractere.

Resultado:

ID	Nome
1	Tadeu
2	Ylane
4	Ewerton
6	Celestino
7	Maria

4.6.2.13. Operador Comparativo “is null” (Valor nulo)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE igual a nulo.

Exemplo:

```
select ID, Nome from Funcion  
where Setor is null;
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Setor seja igual a nulo.

Resultado:

ID	Nome
7	Maria

4.6.2.14. Operador Comparativo “is not null” (Valor não nulo)

Retorna apenas os registros que tenham o valor do atributo citado na cláusula WHERE diferente de nulo.

Exemplo:

```
select ID, Nome from Funcion  
where Setor is not null;
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Setor diferente de nulo.

Resultado:

ID	Nome
1	Tadeu
2	Ylane
3	Julian
4	Ewerton
5	João
6	Celestino

Os exemplos sobre operadores comparativos apresentados neste capítulo podem ser usados por atributos de qualquer tipo (CHAR, VARCHAR, INTEGER, NUMBER, DATE, etc.), a exceção vai para o operador comparativo LIKE e seu complemento NOT LIKE que só podem ser usados por atributos do tipo CHAR, VARCHAR e equivalentes de cada SGBDR.

Até aqui foram vistos os comandos DML, além da primeira parte dos comandos DDL e do comando SELECT. Com esses comandos já fica visível o poder da SQL e o porquê dela ter se tornado uma linguagem padrão os bancos de dados relacionais. No próximo capítulo será apresentada a segunda parte do comando SELECT, e conceitos mais complexos disponíveis na DDL.

5

Capítulo

SQL Avançada

Objetivo

- O comando SELECT vai além do uso da cláusula WHERE, vários sub-comandos e funções podem ser usadas em conjunto com o comando SELECT, transformando-o num comando com uma ortogonalidade elevada. Este capítulo proporciona uma visão mais aprofundada no comando SELECT, sendo finalizado com conceitos e comandos da DDL mais avançados.

Introdução

Neste capítulo serão apresentadas características mais complexas do comando SELECT. Também serão vislumbrados conceitos disponíveis na DDL que não foram oferecidos no capítulo anterior.

1. Comando SELECT – Parte 2

O comando SELECT é muito mais que consultas simples utilizando a cláusula WHERE. Nesta segunda parte será apresentado como executar a cláusula WHERE com condições complexa, além de mostrar:

1. O motivo para efetuar junções de tabelas e como fazê-lo;
2. Como efetuar uniões de tabelas e qual a diferença entre uma união e uma junção;
3. Funções básicas contidas na SQL;
4. Etc.

1.1. Cláusula WHERE com Condições Complexas

Para filtrar registros que requerem condições complexas é utilizada a cláusula WHERE em conjunto com os operadores comparativos e lógicos.

Sintaxe:

```
select Atributo [ {, Atributo} ] from Tabela  
[ where Condição [ {Operador_Lógico Condição} ] ];
```

1.1.1. Operador Lógico “and” (E)

Retorna apenas os registros que atendam todas as condições citados na cláusula WHERE.

Exemplo:

```
select ID, Nome from Funcion
where ID_Setor is not null and ID = 1;
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo ID_Setor seja igual a nulo e o atributo ID seja igual a 1.

Resultado:

```
ID      Nome
1       Tadeu
```

A tabela 5.1 apresenta a tabela verdade deste operador lógico.

Tabela 5.1

TABELA VERDADE DO OPERADOR LÓGICO AND.		
1ª Condição	2ª Condição	Resultado
Falso	Falso	Falso
Falso	Verdadeiro	Falso
Verdadeiro	Falso	Falso
Verdadeiro	Verdadeiro	Verdadeiro

1.1.2. Operador Lógico “or” (OU)

Retorna apenas os registros que atendam a pelo menos uma das condições citados na cláusula WHERE.

Exemplo:

```
select ID, Nome from Funcion
where Nome = 'Tadeu' or ID_Setor = 3;
```

O resultado deste exemplo trará todos os registros da tabela Funcion onde o atributo Nome seja igual a “Tadeu” ou o atributo ID_Setor seja igual a 3.

Resultado:

```
ID      Nome
1       Tadeu
6       Celestino
```

A tabela 5.2 apresenta a tabela verdade deste operador lógico.

Tabela 5.2

TABELA VERDADE DO OPERADOR LÓGICO OR		
1ª Condição	2ª Condição	Resultado
Falso	Falso	Falso
Falso	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Verdadeiro	Verdadeiro	Verdadeiro

1.1.3. Regras de Precedência

As condições complexas seguem algumas regras de precedência, essas regras estão descritas na tabela 5.3.

Tabela 5.3

ORDEM DE PRECEDÊNCIA	
Ordem	Regra
1	Expressões entre parênteses “(...)”
2	Todos os operadores de comparação “=,<>,>,<,<=,IN...”
3	Operador lógico “AND”
4	Operador lógico “OR”

Caso duas condições estejam na mesma ordem de precedência, terá maior precedência a que estiver mais próxima da cláusula WHERE.

1.2. Cláusula ORDER BY

Através da cláusula ORDER BY é possível mostrar os registros de uma consulta ordenados por um ou mais atributos. A ordenação tanto pode ser ascendente como descendente.

Sintaxe:

```
select Atributo [ {, Atributo} ] from Tabela
[ where Condição [ {Operador_Lógico Condição} ] ]
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]}
]];
```

1.2.1. Modo de Ordenação Ascendente - ASC

A palavra reservada ASC ordena o atributo que a precede de modo ascendente. Este modo de ordenação é o padrão, logo caso não se informe o tipo de ordenação após o atributo a ordenação será ascendente.

Exemplo₁:

```
select ID, Nome from Funcion  
order by Nome asc;
```

O resultado deste exemplo trará todos os registros da tabela Funcion ordenados pelo atributo Nome de forma ascendente.

Resultado₁:

ID	Nome
6	Celestino
4	Ewerton
5	João
3	Julian
7	Maria
1	Tadeu
2	Ylane

Exemplo₂:

```
select ID, Nome from Funcion  
order by Nome;
```

O resultado deste exemplo é igual ao anterior, uma vez que a ordenação ascendente é padrão, logo a palavra reservada ASC pode ser suprimida.

Exemplo₃:

```
select ID, Nome from Funcion  
order by 2;
```

Neste exemplo, o nome do atributo Nome da tabela Funcion foi trocado pela posição que o mesmo atributo aparecerá na lista do comando SELECT. As boas práticas recomendam o não uso desta abordagem, pois no caso de mudança nos atributos do comando SELECT, a cláusula ORDER BY deve ser revisada. Outra desvantagem é a impossibilidade de poder ordenar os registros por um atributo que não estejam na lista do comando SELECT.

1.2.2. Modo de Ordenação Descendente - DESC

A palavra reservada DESC ordena o atributo que a precede de modo descendente.

Exemplo:

```
select ID, Nome from Funcion  
order by Nome desc;
```

O resultado deste exemplo trará todos os registros da tabela Funcion ordenados pelo atributo Nome de forma descendente.

Resultado:

ID	Nome
2	Ylane
1	Tadeu
7	Maria
3	Julian
5	João
4	Ewerton
6	Celestino

1.3. Comando JOIN

Ao normalizar o banco de dados (Leia 2.3. Normalização), as informações de um tipo entidade do modelo conceitual podem ser distribuídas por dois ou mais tipos entidade do modelo lógico, vale lembrar que cada tipo entidade do modelo lógico irá se transformar numa tabela do modelo físico (Ler capítulo 3).

Mas ao consultar as informações, algumas vezes é necessário juntar os dados que foram distribuídos pelas tabelas no momento da normalização. O comando JOIN permite trazer os dados de duas ou mais tabelas no resultado de um único SELECT.

Sintaxe:

```
select Atributo [ {, Atributo} ] from Tabela  
[ where Condição [ {Operador_Lógico Condição} ] ]  
[ [Cláusula_Junção] join Tabela_Junção on Condição_Junção]  
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]}  
]];
```

1.3.1. Cláusula INNER

Com a cláusula INNER só serão mostrados os registros que tenham referência nas duas tabelas.

Exemplo₁:

```
select Funcion.ID, Funcion.Nome, Setor.Setor from Funcion  
inner join Setor on (Setor.ID = Funcion.ID_Setor);
```

O resultado deste exemplo trará os atributos ID e Nome de todos os registros da tabela Funcion e o atributo Setor de todos os registros da tabela Setor onde o atributo ID da tabela Setor seja igual ao atributo ID_Setor da tabela Funcion.

No resultado, o registro com o atributo ID igual a 7 da tabela Funcion não aparecerá, pois o atributo ID_Setor é igual a nulo, conseqüentemente não tem referência na tabela Setor.

Também não aparecerá no resultado o registro com o atributo ID igual a 4 da tabela Setor, pois o atributo ID igual 4 não tem referência na tabela Funcion.

Resultado₁:

ID	Nome	Setor
1	Tadeu	Desenvolvimento
2	Ylane	Manutenção
3	Julian	Desenvolvimento
4	Ewerton	Desenvolvimento
5	João	Manutenção
6	Celestino	Financeiro

Este modo de junção é padrão, logo a cláusula INNER pode ser suprimida, como mostrado no próximo exemplo. O resultado do exemplo2 é igual ao resultado do exemplo1.

Exemplo₂:

```
select Funcion.ID, Funcion.Nome, Setor.Setor from Funcion  
join Setor on (Setor.ID = Funcion.ID_Setor);
```

Uma boa prática é dar um apelido as tabelas envolvidas na junção, pois dessa maneira quando se for referenciar a um atributo basta informar o apelido no lugar do nome da tabela. O apelido deve ser informado logo após a tabela, separado apenas por espaço.

Os dois próximos exemplos são semelhantes ao exemplo anterior, a diferença está no uso dos apelidos.

Exemplo₃:

```
select fnc.ID, fnc.Nome, st.Setor from Funcion fnc  
join Setor st on (st.ID = fnc.ID_Setor);
```

Exemplo₄:

```
select a.ID, a.Nome, b.Setor from Funcion a  
join Setor b on (b.ID = a.ID_Setor);
```

1.3.2. Cláusula OUTER

Com a cláusula OUTER serão mostrados os registros que tenham ou não referência nas duas tabelas.

1.3.2.1. Palavra Reservada LEFT

A palavra reservada LEFT antes da cláusula OUTER indica que todos os registros da tabela a esquerda do JOIN (primeira tabela da junção) serão mostrados, independente dela ter ou não referência na tabela da direita do JOIN (segunda tabela da junção).

Exemplo:

```
select a.ID, a.Nome, b.Setor from Funcion a  
left outer join Setor b on (b.ID = a.ID_Setor);
```

O resultado deste exemplo trará os atributos ID e Nome de todos os registros da tabela Funcion e o atributo Setor de todos os registros da tabela Setor onde o atributo ID da tabela Setor seja igual ao atributo ID_Setor da tabela Funcion.

No resultado, o registro com o atributo ID igual a 7 da tabela Funcion aparecerá, pois mesmo sem o seu atributo ID_Setor ter uma referência na tabela Setor, a palavra reservada LEFT permite sua aparição.

Mas não aparecerá no resultado o registro com o atributo ID igual a 4 da tabela Setor, pois o atributo ID igual 4 não tem referência na tabela Funcion.

Resultado:

ID	Nome	Setor
1	Tadeu	Desenvolvimento
2	Ylane	Manutenção
3	Julian	Desenvolvimento
4	Ewerton	Desenvolvimento
5	João	Manutenção
6	Celestino	Financeiro
7	Maria	null

1.3.2.2. Palavra Reservada RIGHT

A palavra reservada RIGHT antes da cláusula OUTER indica que todos os registros da tabela a direita do JOIN (segunda tabela da junção) serão mostrados, independente dela ter ou não referência na tabela da esquerda do JOIN (primeira tabela da junção).

Exemplo:

```
select a.ID, a.Nome, b.Setor from Funcion a
right outer join Setor b on (b.ID = a.ID_Setor);
```

O resultado deste exemplo trará os atributos ID e Nome de todos os registros da tabela Funcion e o atributo Setor de todos os registros da tabela Setor onde o atributo ID da tabela Setor seja igual ao atributo ID_Setor da tabela Funcion.

No resultado, o registro com o atributo ID igual a 4 da tabela Setor aparecerá, pois mesmo sem o seu atributo ID ter uma referência na tabela Funcion, a palavra reservada RIGHT permite sua aparição.

Mas não aparecerá no resultado o registro com o atributo ID igual a 7 da tabela Funcion, pois o atributo ID_Setor é igual a nulo, consequentemente não tem referência na tabela Setor.

Resultado:

ID	Nome	Setor
1	Tadeu	Desenvolvimento
2	Ylane	Manutenção
3	Julian	Desenvolvimento
4	Ewerton	Desenvolvimento
5	João	Manutenção
6	Celestino	Financeiro
null	null	Vendas

1.3.2.3. Palavra Reservada FULL

A palavra reservada FULL antes da cláusula OUTER indica que todos os registros da tabela tabelas envolvidas no JOIN serão mostradas independente do registro ter ou não referência nas duas tabelas.

Exemplo:

```
select a.ID, a.Nome, b.Setor from Funcion a  
full outer join Setor b on (b.ID = a.ID_Setor);
```

O resultado deste exemplo trará os atributos ID e Nome de todos os registros da tabela Funcion e o atributo Setor de todos os registros da tabela Setor onde o atributo ID da tabela Setor seja igual ao atributo ID_Setor da tabela Funcion.

No resultado, o registro com o atributo ID igual a 7 da tabela Funcion aparecerá, pois mesmo sem o seu atributo ID_Setor ter uma referência na tabela Setor, a palavra reservada FULL permite sua aparição.

No resultado também aparecerá o registro com o atributo ID igual a 4 da tabela Setor, pois mesmo sem o seu atributo ID ter uma referência na tabela Funcion, a palavra reservada FULL permite sua aparição.

Resultado:

ID	Nome	Setor
1	Tadeu	Desenvolvimento
2	Ylane	Manutenção
3	Julian	Desenvolvimento
4	Ewerton	Desenvolvimento
5	João	Manutenção
6	Celestino	Financeiro
7	Maria	null
null	null	Vendas

1.4. Comando UNION

O comando UNION permite trazer os registros de duas ou mais tabelas no resultado de um único SELECT.

Para quem está vendo pela primeira vez os comandos UNION e JOIN, a explicação dos dois parecem semelhantes, mas os dois comandos são totalmente diferentes.

Enquanto no comando JOIN cada registro mostrado é composto por atributos de duas ou mais tabelas, no comando UNION cada registro mostrado é composto por atributos de apenas uma única tabela, mas os registros mostrados pertencem a duas ou mais tabelas.

Sintaxe:

```
select Atributo [ {, Atributo} ] from Tabela
[ where Condição [ {Operador_Lógico Condição} ] ]
[ [Cláusula_Junção] join Tabela_Junção on Condição_Junção]
[
union [all]
select Atributo [ {, Atributo} ] from Tabela
[ where Condição [ {Operador_Lógico Condição} ] ]
[ [Cláusula_Junção] join Tabela_Junção on Condição_Junção]
]
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]}
]];
```

Exemplo:

```
select Nome from Funcion
union
select Nome from Cliente;
```

O resultado deste exemplo trará o atributo Nome de todos os registros da tabela Funcion unido com os atributos Nome de todos os registros da tabela Cliente. O nome “João” e “Maria” existem nas duas tabelas, mas cada um só aparece uma vez no resultado.

Resultado:

```
Nome
Adriana
Celestino
Eduarda
Ewerton
Francisco
José
```

João
Julian
Maria
Tadeu
Ylane

1.4.1. Cláusula ALL

Por padrão os registros duplicados são eliminados do resultado, para mostrar todos os registros, idênticos ou não, deve-se utilizar a cláusula ALL.

Exemplo:

```
select Nome from Funcion  
union all  
select Nome from Cliente;
```

O resultado deste exemplo trará o atributo Nome de todos os registros da tabela Funcion unido com os atributos Nome de todos os registros da tabela Cliente. O nome “João” e “Maria” existem nas duas tabelas, e neste exemplo aparecem duas vezes no resultado.

Resultado:

Nome
Tadeu
Ylane
Julian
Ewerton
João
Celestino
Maria
Francisco
José
Maria
Adriana
João
Eduardo

1.5. Funções Básicas

A SQL tem funções básicas que ajudam no resgates das informações relacionadas aos dados armazenados.

Sintaxe:

```
select Função(Parâmetro) [{, Função(Parâmetro)}] from Tabela  
[ where Condição [ {Operador_Lógico Condição} ] ];
```

1.5.1. Função AVG

A função AVG retorna a média aritmética do atributo passado como parâmetro.

Exemplo₁:

```
select avg(Salario) from Funcion;
```

O resultado deste exemplo trará a média aritmética dos valores do atributo Salario da tabela Funcion.

Resultado₁:

```
AVG  
1071.4285714285714286
```

É possível mudar o rótulo de um atributo que aparecerá na lista do comando SELECT, para isto basta acrescentar após o atributo a palavra reservada AS seguida do nome que irá substituir o nome do atributo.

Exemplo₂:

```
select avg(Salario) as Media from Funcion;
```

O resultado deste exemplo trará a média aritmética dos valores do atributo Salario da tabela Funcion.

Resultado₂:

```
Media  
1071.4285714285714286
```

1.5.2. Função MAX

A função MAX retorna o maior valor do atributo passado como parâmetro.

Exemplo:

```
select max(Salario) as Maior_Salario from Funcion;
```

O resultado deste exemplo trará o maior valor inserido no atributo Salario da tabela Funcion.

Resultado:

```
Maior_Salario  
1500.00
```

1.5.3. Função MIN

A função MIN retorna o menor valor do atributo passado como parâmetro.

Exemplo:

```
select min(Salario) as Menor_Salario from Funcion;
```

O resultado deste exemplo trará o menor valor inserido no atributo Salario da tabela Funcion.

Resultado:

```
Menor_Salario  
500.00
```

1.5.4. Função SUM

A função SUM retorna o somatório do valor do atributo passado como parâmetro.

Exemplo:

```
select sum(Salario) as Soma_Salario from Funcion;
```

O resultado deste exemplo trará a soma dos valores inseridos no atributo Salario da tabela Funcion.

Resultado:

Soma_Salario

7500.00

1.5.5. Função COUNT

A função COUNT retorna a quantidade de registros não nulos do atributo passado como parâmetro.

Exemplo₁:

```
select count(Salario) as Quant_Salario from Funcion;
```

O resultado deste exemplo trará a quantidade de registro que têm um valor informado no atributo Salario da tabela Funcion.

Resultado₁:

Quant_Salario

7

Exemplo₂:

```
select count(ID_Setor) as Quant_Setor from Funcion;
```

O resultado deste exemplo trará a quantidade de registro que têm um valor informado no atributo ID_Setor da tabela Funcion.

Resultado₂:

Quant_Setor

6

1.6. Cláusula GROUP BY

Para recuperar o resultado de uma ou mais funções agrupadas por um ou mais atributos deve-se usar a cláusula GROUP BY.

Sintaxe:

```
select Função(Parâmetro) [{, Função(Parâmetro)}] [{, Atributo}] from Ta-  
bela
```

```
[ where Condição [ {Operador_Lógico Condição} ] ]
[ group by Atributo [{, Atributo}] ]
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]}
]];
```

Exemplo:

```
select sum(salario) as Soma_Salario, ID_Setor from Funcion
group by ID_Setor;
```

O resultado deste exemplo trará a soma dos valores inseridos no atributo Salario da tabela Funcion agrupados pelo atributo ID_Setor. Todos os atributos que forem informados antes da palavra reservada FROM devem ser declarados na cláusula GROUP BY.

Resultado:

Soma_Salario	ID_Setor
500.00	null
1500.00	3
2000.00	2
3500.00	1

Este resultado informa que o valor total da folha de pagamento do setor 1 é de R\$ 3.500,00, enquanto do setor 2 é de R\$ 2.000,00. O setor 3 tem uma folha de pagamento no valor total de R\$ 1.500,00 . Já a folha de pagamentos dos funcionários que não estão lotados em nenhum setor totaliza R\$ 500,00.

1.6.1. Cláusula HAVING

Caso seja necessário filtrar o resultado de uma ou mais funções agrupadas pela cláusula GROUP BY, deve-se usar a cláusula HAVING.

Sintaxe:

```
select Função(Parâmetro) [{, Função(Parâmetro)}] [{, Atributo}] from Tabela
[ where Condição [ {Operador_Lógico Condição} ] ]
[ group by Atributo [{, Atributo}] ]
[ having Condição_Função ]
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]}
]];
```

Exemplo:

```
select sum(salario) as Soma_Salario, ID_Setor from Funcion
group by ID_Setor
having sum(salario) > 1500
```

O resultado deste exemplo trará a soma dos valores inseridos no atributo Salario da tabela Funcion agrupados pelo atributo ID_Setor, onde a soma dos valores inseridos no atributo Salario seja maior que R\$ 1.500,00.

Resultado:

Soma_Salario	ID_Setor
2000.00	2
3500.00	1

1.7. Cláusula DISTINCT

Para filtrar os valores duplicados de um atributo recuperado pelo comando SELECT deve-se usar a cláusula DISTINCT.

Sintaxe:

```
select distinct(Atributo) [ {, Atributo} ] from Tabela;
[ where Condição [ {Operador_Lógico Condição} ] ]
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]} ] ];
```

Exemplo:

```
select distinct(ID_Setor) from Funcion
```

O resultado deste exemplo trará o valor do atributo ID_Setor da tabela Funcion filtrando os valores duplicados.

Resultado:

ID_Setor
null
1
2
3

1.8. Operadores de Manipulação

A SQL permite operações de manipulação sobre os atributos que aparecerão na lista do SELECT através dos operadores de manipulação.

Sintaxe:

```
select Operação_Manipulação [ {, Atributo} ] from Tabela;  
[ where Condição [ {Operador_Lógico Condição} ] ]  
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]}  
]];
```

1.8.1. Operador de Manipulação “||” (Concatenação)

Para combinar duas ou mais cadeias de caracteres (atributos ou constantes) e apresentar seu resultado como um atributo da lista do SELECT deve-se usar o operador “||”;

Exemplo:

```
select Nome || ' ' || Sobrenome as Nome_Completo from Cliente;
```

O resultado deste exemplo trará os atributos Nome e Sobrenome da tabela Cliente unidos como se fossem um único atributo. Entre os dois atributos foi inserido um espaço em branco para facilitar a leitura.

Resultado:

```
Nome_Completo  
Francisco Silva  
José Lima  
Maria Silva  
Adriana Ferreira  
João Oliveira  
Eduarda Souza
```

1.8.2. Operador de Manipulação “+” (Adição)

Para mostrar a soma de dois ou mais valores (atributos ou constantes) e apresenta seu resultado como um atributo da lista do SELECT deve-se usar o operador “+”;

Exemplo:

```
select Nome, (Salario + 200) as Salario_Atual from Funcion;
```

O resultado deste exemplo trará os atributos Nome e Salario da tabela Funcion, mas o atributo Salario terá o seu valor adicionado em R\$ 200,00.

Resultado:

Nome	Salario_Atual
Tadeu	1700,00
Ylane	1400,00
Julian	1200,00
Ewerton	1200,00
João	1000,00
Celestino	1700,00
Maria	700,00

1.8.3. Operador de Manipulação “-” (Subtração)

Para mostrar a subtração de dois ou mais valores (atributos ou constantes) e apresenta seu resultado como um atributo da lista do SELECT deve-se usar o operador “-”;

Exemplo:

```
select Nome, (Salario - 200) as Salario_Atual from Funcion;
```

O resultado deste exemplo trará os atributos Nome e Salario da tabela Funcion, mas o atributo Salario terá o seu valor subtraído em R\$ 200,00.

Resultado:

Nome	Salario_Atual
Tadeu	1300,00
Ylane	1000,00
Julian	800,00
Ewerton	800,00
João	600,00
Celestino	1300,00
Maria	300,00

1.8.4. Operador de Manipulação "*" (Multiplicação)

Para mostrar a multiplicação de dois ou mais valores (atributos ou constantes) e apresenta seu resultado como um atributo da lista do SELECT deve-se usar o operador "*";

Exemplo:

```
select Nome, (Salario * 1.5) as Salario_Atual from Funcion;
```

O resultado deste exemplo trará os atributos Nome e Salario da tabela Funcion, mas o atributo Salario terá o seu valor multiplicado por 1,5, ou seja, o atributo Salario terá um aumento de 50%.

Resultado:

Nome	Salario_Atual
Tadeu	2250.00
Ylane	1800.00
Julian	1500.00
Ewerton	1500.00
João	1200.00
Celestino	2250.00
Maria	750.00

1.8.5. Operador de Manipulação "/" (Divisão)

Para mostrar a divisão entre dois valores (atributos ou constantes) e apresentar seu resultado como um atributo da lista do SELECT deve-se usar o operador "/";

Exemplo:

```
select Nome, (Salario / 2) as Salario_Quinzena from Funcion;
```

O resultado deste exemplo trará os atributos Nome e Salario da tabela Funcion, mas o atributo Salario terá o seu valor dividido por 2.

Resultado:

Nome	Salario_Quinzena
Tadeu	750.00
Ylane	600.00
Julian	500.00
Ewerton	500.00
João	400.00
Celestino	750.00
Maria	250.00

1.9. Nested Queries

É possível restringir os dados mostrados em uma consulta principal baseados nos resultados de uma sub-consulta. Esse processo é chamado de NESTED QUERIES (Ninhos de Pesquisa).

Sintaxe:

```
select Atributo [ {, Atributo} ] from Tabela;
[ where Atributo Operador_Comparativo
  [ select Atributo_Ninho from Tabela_Ninho [where Condição_Ninho] ]
[ order by Atributo [Modo_Ordenação] [ {, Atributo [Modo_Ordenação]}
]]];
```

Exemplo₁:

```
select ID, Nome from Cliente
where ID in ( select ID_Cliente from Pedido where Valor >= 2000 );
```

O resultado deste exemplo trará os atributos ID e Nome da tabela Cliente onde o atributo ID seja igual a um dos atributos ID_Cliente retornado pela sub-consulta feita na tabela Pedido.

Resultado₁:

ID	Nome
4	Adriana
5	João

Exemplo₂:

```
select ID, Nome from Funcion
where ID_Setor = ( select max(ID) - 1 from Setor );
```

O resultado deste exemplo trará os atributos ID e Nome da tabela Funcion onde o atributo ID seja igual ao maior valor do atributo ID da tabela Setor menos 1.

Resultado₂:

ID	Nome
6	Celestino

2. DDL – Parte 2

Na seção 4.4 vimos os comando básicos da DDL, nesta seção iremos um pouco mais além, abordando views, stored procedures, triggers e domains. A sintaxe usada nesta seção é direcionada para o PostgreSQL, podendo não funcionar em outros SGBDR, uma vez que esta é a parte da SQL que mais muda de um SGBDR para outro.

2.1. VIEW

Uma VIEW funciona como um comando SELECT salvo dentro do banco de dados, assim como acontece com as tabelas e os dados armazenados nas tabelas.

Sintaxe:

```
create view Visão [ ( Atributo [ {,Atributo} ] ) ]
as Comando_Select;
```

Exemplo de Criação:

```
create view Cliente_Pedido (ID, NomeCompleto, SomaPedido)
as
select a.ID, (a.Nome || ' ' || a.Sobrenome), sum(b.Valor) from Cliente a
join Pedido b on (b.ID_Cliente = a.ID)
group by a.ID, a.Nome, a.Sobrenome;
```

Este exemplo cria uma VIEW com o nome de Cliente_Pedido. A lista de atributos que esta VIEW retornará será ID, NomeCompleto, SomaPedido.

O comando SELECT desta VIEW irá fazer uma junção entre as tabelas Cliente e Pedido. Esta junção trará a soma dos valores do atributo Valor da tabela Pedido agrupados pelos atributos ID, Nome e Sobrenome da tabela Cliente. Os atributos Nome e Sobrenome da tabela Cliente foram concatenados com um espaço em branco.

Ao passar a lista de atributos da VIEW (ID, NomeCompleto, SomaPedido) no momento da criação, estamos explicitando o nome dos atributos que a VIEW retornará.

Exemplo de Uso:

```
select * from Cliente_Pedido
where SomaPedido > 2000
order by NomeCompleto;
```

O resultado deste exemplo trará todos os atributos da visão Cliente_Pedido onde o atributo SomaPedido seja maior que 2000 e ordenado pelo atributo NomeCompleto.

Resultado:

ID	NomeCompleto	SomaPedido
5	João Oliveira	2500.00
2	José Lima	3500.00

Com estes dois exemplos pode-se afirmar que a VIEW:

- Funciona como um comando SELECT salvo no banco de dados;
- Simplifica o uso dos comandos SELECTs complexos, pois permite aplicar novos comandos SELECTs sobre ela;
- É vista pelo usuário como uma tabela real do banco de dados, quando na realidade é uma tabela virtual;
- Possibilita mostrar ao usuário uma versão personalizada das tabelas do banco de dados.

2.2. STORE PROCEDURE

Uma STORE PROCEDURE funciona como uma função definida pelo usuário salva dentro do banco de dados. A STORE PROCEDURE também suporta a declaração de variáveis, estruturas condicionais, estruturas de repetição, etc.

Tem como vantagem a velocidade, uma vez que executado dentro do banco de dados. E como desvantagem está o aumento na utilização dos recursos do servidor de banco de dados.

No PostgreSQL STORE PROCEDURE são chamadas simplesmente de FUNCTION.

Sintaxe:

```
create function Função ( Tipo_Entrada [ {, Tipo_Entrada} ] )
[returns Tipo_Returno]
as $$ Instruções_Função $$
language Linguagem_Usada;
```

Exemplo de Criação:

```
create function TipoSalarioFuncion( integer ) returns varchar as
$$
declare
    p_id integer;
    linha numeric(15, 2);
    retorno varchar;
begin
    p_id := $1;
    select Salario from Funcion where id = p_id into linha;
    if ( linha < 1000 ) then
        retorno := 'Salário menor que R$ 1000,00';
    elseif ( linha = 1000 ) then
        retorno := 'Salário igual a R$ 1000,00';
    else
        retorno := 'Salário maior que R$ 1000,00';
    end if;
    return retorno;
end;
$$
language plpgsql;
```

Caso a linguagem procedural padrão *PLPGSQL* não seja reconhecido, deve-se cria-la através do comando **CREATE LANGUAGE PLPGSQL;**

Este exemplo cria uma STORE PROCEDURE com o nome de Tipo-SalarioFuncion, que recebe como parâmetro de entrada um tipo INTEGER e retorna um valor do tipo VARCHAR.

No corpo são declaradas as variáveis `p_id` do tipo `INTEGER`, linha do tipo `NUMERIC` e retorno do tipo `VARCHAR`.

A variável `p_id` recebe o primeiro e único parâmetro de entrada (`$1`), esta variável é usada para regatar o valor do atributo `Salario` da tabela `Funcion` onde o atributo `ID` seja igual a `p_id`.

A variável `linha` recebe o resultado do comando `SELECT`, em seguida é verificado seu valor para atribuir o tipo de salário correto a variável `retorno`, que é passada como retorno da `STORE PROCEDURE`.

A última linha informa a linguagem que esta sendo usada pela `STORE PROCEDURE`. As linguagens padrões do PostgreSQL são **INTERNAL**, **C**, **SQL** e **PLPGSQL**.

Exemplo de Uso₁:

```
select TipoSalarioFuncion(1);
```

O resultado deste exemplo trará o tipo de salário do registro da tabela `Funcion` com o atributo `ID` igual a 1.

Resultado₁:

```
TipoSalarioFuncion
Salário maior que R$ 1000,00
```

Exemplo de Uso₂:

```
select ID, Nome, Salario, TipoSalarioFuncion(ID) from Funcion
```

O resultado deste exemplo trará os atributos `ID`, `Nome` e `Salario` de todos os registros da tabela `Funcion`, com os seus respectivos tipo de salário.

Resultado₂:

ID	Nome	Salario	TipoSalarioFuncion
1	Tadeu	1500,00	Salário maior que R\$ 1000,00
2	Ylane	1200,00	Salário maior que R\$ 1000,00
3	Julian	1000,00	Salário igual a R\$ 1000,00
4	Ewerton	1000,00	Salário igual a R\$ 1000,00
5	João	800,00	Salário menor que R\$ 1000,00
6	Celestino	1500,00	Salário maior que R\$ 1000,00
7	Maria	500,00	Salário menor que R\$ 1000,00

2.3. TRIGGER

Um TRIGGER é um gatilho disparado automaticamente pelo SGBDR quando um comando INSERT, UPDATE ou DELETE é executado numa tabela do banco de dados. No PostgreSQL uma STORE PROCEDURE deve ser criada exclusivamente para se utilizar o TRIGGER.

Sintaxe:

```
create trigger Gatilho { {before | after} Evento [ {or Evento} ] } on Tabela  
[ for [each] {row | statement} ] execute procedure Store_Procedure;
```

As palavras reservadas BEFORE e AFTER indicam se o TRIGGER vai ser disparado antes ou depois do evento, respectivamente. Já o evento pode ser um comando INSERT, UPDATE ou DELETE.

Se na criação do TRIGGER for informado o FOR EACH ROW, o gatilho será disparado uma vez para cada linha modificada. Informando o FOR EACH STATEMENT, o gatilho só será disparado uma vez, independente da quantidade de linhas modificadas.

Exemplo de Criação da STORE PROCEDURE usada pelo TRIGGER:

```
create or replace function MaxIDFuncion() returns trigger as  
$$  
declare  
    max_ID integer;  
begin  
    select max(id) from Funcion into max_ID;  
    new.id := max_ID + 1;  
    return new;  
end;  
$$  
language plpgsql;
```

Neste exemplo foi criado a STORE PROCEDURE que será usada pelo TRIGGER. O exemplo executa a função MAX sobre o atributo ID da tabela Funcion e atribui o resultado a variável max_ID. A variável max_ID acrescido de um é atribuído ao novo ID (new.id) do evento. As variáveis NEW e OLD são criadas automaticamente.

A variável NEW mantém os novos valores de todos os atributos do registro que está sendo enviada ao banco de dados, só pode ser usada em conjunto com os comandos INSERT e UPDATE.

A variável OLD é o oposto, mantém os antigos valores de todos os atributos do registro que está sendo enviada ao banco de dados, só pode ser usada em conjunto com os comandos UPDATE e DELETE.

Exemplo de Criação do TRIGGER:

```
create trigger Trg_Funcion before insert on Funcion
for each row execute procedure MaxIDFuncion();
```

Neste exemplo foi criado um TRIGGER com o nome de Trg_Funcion para a tabela Funcion. Ele será disparado antes do comando INSERT e executará a STORE PROCEDURE MaxIDFuncion para cada linha modificada.

Exemplo de Uso:

```
insert into Funcion (Nome, Salario, ID_Setor) values ('Jesus', 2000, 4);
select * from Funcion;
```

O primeiro comando insere um novo registro na tabela Funcion, mas o mesmo não informa o atributo ID desse novo registro. A inserção dispara o TRIGGER Trg_Funcion que executa a STORE PROCEDURE MaxIDFuncion.

A STORE PROCEDURE MaxIDFuncion captura o maior valor do atributo ID da tabela Funcion, adiciona em 1 e o atribui a variável NEW.ID, o novo registro terá o valor do seu atributo ID igual a variável NEW.ID.

O segundo comando trará todos os atributos de todos os registros da tabela Funcion já com o novo registro devidamente cadastrado.

Resultado:

ID	Nome	Salario	ID_Setor
1	Tadeu	1500,00	1
2	Ylane	1200,00	2
3	Julian	1000,00	1
4	Ewerton	1000,00	1
5	João	800,00	2
6	Celestino	1500,00	3
7	Maria	500,00	null
8	Jesus	2000.00	4

2.4. DOMAIN

Um DOMAIN ou domínio é um tipo de dado definidos pelo usuário com a finalidade de reaproveitamento. O DOMAIN padroniza e centraliza os tipos de dados utilizados pelas tabelas do banco de dados.

Sintaxe:

```
create domain Nome_Domain [as] Tipo_Dado  
[default Valor_Padrão] [Restrição] [check(Restrição)];
```

Exemplo:

```
create domain DM_VALOR as numeric(15,2)  
default 0 not null check (value >= 0);
```

Neste exemplo foi criado um DOMAIN com o nome de DM_VALOR do tipo Numeric(15,2), onde seu valor padrão será zero, ele não pode ser nulo e nunca aceitará que seja inserido nele um valor inferior a zero.

Exemplo de Uso:

```
create table Conta (  
    ID integer not null primary key,  
    Agencia varchar(5),  
    Saldo numeric(15,2) default 0 not null,  
    LimiteConta DM_VALOR,  
    LimiteEmprestimo DM_VALOR  
);
```

Neste exemplo foi criada a tabela Conta. O atributo ID é do tipo Integer não pode ser nulo e é uma chave primária, enquanto o atributo Agencia é do tipo Varchar(5). O atributo Saldo é do tipo Numeric(15,2), seu valor padrão é zero e não pode ser nulo.

Os atributos LimiteConta e LimiteEmprestimo são do tipo DM_VALOR. Isto está dizendo implicitamente ao SGBDR que os atributos são do tipo Numeric(15,2), seu valor padrão é zero, não podem ser nulo e nunca deverá ser incluído neles valores negativos.

Referências



- [DEITEL, 2010] DEITEL, Paul; DEITEL, Harvey. **Java: Como Programar**. 8. ed. São Paulo, Pearson Prentice Hall, 2010.
- [ELMASRI e NAVATHE, 2005] ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados**. 4. ed. São Paulo, Pearson Addison Wesley, 2005.
- [HEUSER, 2001] HEUSER, Carlos Alberto. **Projeto de Banco de Dados**. 4. ed. Porto Alegre, Sagra Luzzatto, 2001.
- [POSTGRESQL, 2011] POSTGRESQL. **Documentação do PostgreSQL 8.2.0**. <http://pgdocptbr.sourceforge.net/pg82/history.html>. Setembro de 2011.
- [RAMAKRISHNAN e GEHRKE, 2008] RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de Banco de Dados**. 2. ed. São Paulo: McGraw-Hill, 2008. 884 p.
- [ROB e CORONEL, 2011] ROB, Peter; CORONEL, Carlos. **Sistema de Banco de Dados**. São Paulo, Cengage Learning, 2011.
- [SOMMERVILLE, 2007] SOMMERVILLE, Ian. **Engenharia de Software**. 8. ed. São Paulo, Pearson Addison Wesley, 2007.

Sobre os autores

Cicero Tadeu Pereira Lima França: É mestre em Computação Aplicada pela UECE, especialista em Engenharia de Software com ênfase em Padrões de Software e especialista em Gestão de Projetos de TI, graduado como tecnólogo em Automática pelo IFCE. Tem experiência na área de Ciência da Computação, atuando principalmente no desenvolvimento de softwares. Professor da Faculdade Leão Sampaio.

Joaquim Celestino Júnior: Possui graduação em Engenharia Eletrônica pela Pontifícia Universidade Católica do Rio de Janeiro (1985), mestrado em Ciência da Computação pela Universidade Federal da Paraíba/Campina Grande(1989), doutorado em Redes de Computadores - Université de Paris VI (Pierre et Marie Curie) (1994) e pós-doutorado em Redes Veiculares pela Columbia University in New York City (2010). É professor adjunto da Universidade Estadual do Ceará (UECE). Tem experiência na área de Ciência da Computação, com ênfase em Teleinformática, atuando principalmente nos seguintes temas: redes de computadores, gerenciamento de redes e segurança.



A não ser que indicado ao contrário a obra **Banco de Dados**, disponível em: <http://educapes.capes.gov.br>, está licenciada com uma licença **Creative Commons Atribuição-Compartilha Igual 4.0 Internacional (CC BY-SA 4.0)**. Mais informações em: <http://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR>. Qualquer parte ou a totalidade do conteúdo desta publicação pode ser reproduzida ou compartilhada. Obra sem fins lucrativos e com distribuição gratuita. O conteúdo do livro publicado é de inteira responsabilidade de seus autores, não representando a posição oficial da EdUECE.



Computação

Fiel a sua missão de interiorizar o ensino superior no estado Ceará, a UECE, como uma instituição que participa do Sistema Universidade Aberta do Brasil, vem ampliando a oferta de cursos de graduação e pós-graduação na modalidade de educação a distância, e gerando experiências e possibilidades inovadoras com uso das novas plataformas tecnológicas decorrentes da popularização da internet, funcionamento do cinturão digital e massificação dos computadores pessoais.

Comprometida com a formação de professores em todos os níveis e a qualificação dos servidores públicos para bem servir ao Estado, os cursos da UAB/UECE atendem aos padrões de qualidade estabelecidos pelos normativos legais do Governo Federal e se articulam com as demandas de desenvolvimento das regiões do Ceará.



UNIVERSIDADE ESTADUAL DO CEARÁ

