Modelagem de Banco de Dados

Paulo Henrique Cayres

A RNP – Rede Nacional de Ensino e Pesquisa – é qualificada como uma Organização Social (OS), sendo ligada ao Ministério da Ciência, Tecnologia e Inovação (MCTI) e responsável pelo Programa Interministerial RNP, que conta com a participação dos ministérios da Educação (MEC), da Saúde (MS) e da Cultura (MinC). Pioneira no acesso à Internet no Brasil, a RNP planeja e mantém a rede Ipê, a rede óptica nacional acadêmica de alto desempenho. Com Pontos de Presença nas 27 unidades da federação, a rede tem mais de 800 instituições conectadas. São aproximadamente 3,5 milhões de usuários usufruindo de uma infraestrutura de redes avançadas para comunicação, computação e experimentação, que contribui para a integração entre o sistema de Ciência e Tecnologia, Educação Superior, Saúde e Cultura.

RNP

Ministério da **Cultura**

Ministério da **Saúde**

Ministério da **Educação**

Ministério da **Ciência, Tecnologia e Inovação**

Modelagem de Banco de Dados

Paulo Henrique Cayres

Modelagem de Banco de Dados

Paulo Henrique Cayres

Rio de Janeiro
Escola Superior de Redes
2015

Rua Lauro Müller, 116 sala 1103
22290-906 Rio de Janeiro, RJ

Diretor Geral
Nelson Simões

Diretor de Serviços e Soluções
José Luiz Ribeiro Filho

Escola Superior de Redes

Coordenação
Luiz Coelho

Edição
Lincoln da Mata

Coordenador Acadêmico da Área de Desenvolvimento de Sistemas
John Lemos Forman

Equipe ESR (em ordem alfabética)
Adriana Pierro, Celia Maciel, Cristiane Oliveira, Derlinéa Miranda, Edson Kowask, Elimária Barbosa, Evellyn Feitosa, Felipe Nascimento, Lourdes Soncin, Luciana Batista, Luiz Carlos Lobato, Renato Duarte e Yve Abel Marcial.

Capa, projeto visual e diagramação
Tecnodesign

Versão
1.0.0

Distribuição
Escola Superior de Redes
Rua Lauro Müller, 116 – sala 1103
22290-906 Rio de Janeiro, RJ
http://esr.rnp.br
info@esr.rnp.br

Dados Internacionais de Catalogação na Publicação (CIP)

C385m Cayres, Paulo Henrique
Modelagem de banco de dados / Paulo Henrique Cayres. – Rio de Janeiro: RNP/ESR, 2015.
182 p. : il. ; 28 cm.

ISBN 978-85-63630-50-6

1. Modelagem de dados. 2. Banco de dados. 3. Sistema de gerenciamento de banco de dados (SGBD). 4. PostgreSQL. I. Titulo.

CDD 005.743

Sumário

Escola Superior de Redes

A Escola Superior de Redes (ESR) é a unidade da Rede Nacional de Ensino e Pesquisa (RNP) responsável pela disseminação do conhecimento em Tecnologias da Informação e Comunicação (TIC). A ESR nasce com a proposta de ser a formadora e disseminadora de competências em TIC para o corpo técnico-administrativo das universidades federais, escolas técnicas e unidades federais de pesquisa. Sua missão fundamental é realizar a capacitação técnica do corpo funcional das organizações usuárias da RNP, para o exercício de competências aplicáveis ao uso eficaz e eficiente das TIC.

A ESR oferece dezenas de cursos distribuídos nas áreas temáticas: Administração e Projeto de Redes, Administração de Sistemas, Segurança, Mídias de Suporte à Colaboração Digital e Governança de TI.

A ESR também participa de diversos projetos de interesse público, como a elaboração e execução de planos de capacitação para formação de multiplicadores para projetos educacionais como: formação no uso da conferência web para a Universidade Aberta do Brasil (UAB), formação do suporte técnico de laboratórios do Proinfo e criação de um conjunto de cartilhas sobre redes sem fio para o programa Um Computador por Aluno (UCA).

A metodologia da ESR

A filosofia pedagógica e a metodologia que orientam os cursos da ESR são baseadas na aprendizagem como construção do conhecimento por meio da resolução de problemas típicos da realidade do profissional em formação. Os resultados obtidos nos cursos de natureza teórico-prática são otimizados, pois o instrutor, auxiliado pelo material didático, atua não apenas como expositor de conceitos e informações, mas principalmente como orientador do aluno na execução de atividades contextualizadas nas situações do cotidiano profissional.

A aprendizagem é entendida como a resposta do aluno ao desafio de situações-problema semelhantes às encontradas na prática profissional, que são superadas por meio de análise, síntese, julgamento, pensamento crítico e construção de hipóteses para a resolução do problema, em abordagem orientada ao desenvolvimento de competências.

Dessa forma, o instrutor tem participação ativa e dialógica como orientador do aluno para as atividades em laboratório. Até mesmo a apresentação da teoria no início da sessão de aprendizagem não é considerada uma simples exposição de conceitos e informações. O instrutor busca incentivar a participação dos alunos continuamente.

As sessões de aprendizagem onde se dão a apresentação dos conteúdos e a realização das atividades práticas têm formato presencial e essencialmente prático, utilizando técnicas de estudo dirigido individual, trabalho em equipe e práticas orientadas para o contexto de atuação do futuro especialista que se pretende formar.

As sessões de aprendizagem desenvolvem-se em três etapas, com predominância de tempo para as atividades práticas, conforme descrição a seguir:

Primeira etapa: apresentação da teoria e esclarecimento de dúvidas (de 60 a 90 minutos).
O instrutor apresenta, de maneira sintética, os conceitos teóricos correspondentes ao tema da sessão de aprendizagem, com auxílio de slides em formato PowerPoint. O instrutor levanta questões sobre o conteúdo dos slides em vez de apenas apresentá-los, convidando a turma à reflexão e participação. Isso evita que as apresentações sejam monótonas e que o aluno se coloque em posição de passividade, o que reduziria a aprendizagem.

Segunda etapa: atividades práticas de aprendizagem (de 120 a 150 minutos).
Esta etapa é a essência dos cursos da ESR. A maioria das atividades dos cursos é assíncrona e realizada em duplas de alunos, que acompanham o ritmo do roteiro de atividades proposto no livro de apoio. Instrutor e monitor circulam entre as duplas para solucionar dúvidas e oferecer explicações complementares.

Terceira etapa: discussão das atividades realizadas (30 minutos).
O instrutor comenta cada atividade, apresentando uma das soluções possíveis para resolvê-la, devendo ater-se àquelas que geram maior dificuldade e polêmica. Os alunos são convidados a comentar as soluções encontradas e o instrutor retoma tópicos que tenham gerado dúvidas, estimulando a participação dos alunos. O instrutor sempre estimula os alunos a encontrarem soluções alternativas às sugeridas por ele e pelos colegas e, caso existam, a comentá-las.

Sobre o curso

Apresenta de forma clara e objetiva o processo de modelagem, criação e manutenção de banco de dados relacionais. Apresenta uma visão geral sobre bancos de dados, bem como conceitos e metodologias para modelagem conceitual, lógica e física de banco de dados relacionais. Aborda as principais características e funcionalidades de um sistema gerenciador de bancos de dados usando como base o PostgreSQL.

Estas práticas e regras, para a correta implementação de projetos de banco de dados relacionais, são ferramentas utilizadas no curso e que se aplicam a qualquer sistema baseado na linguagem SQL.

Cada sessão apresenta um conjunto de exemplos e atividades práticas que permitem a prática das habilidades apresentadas.

A quem se destina

Pessoas interessadas em trabalhar com bancos de dados, se familiarizando com técnicas de modelagem, projeto e manipulação de banco de dados relacionais. É uma porta de entrada para quem quer se aprofundar no assunto e considera trabalhar como analista de dados, administrador de bancos de dados ou também para programadores interessados em conhecer o processo de modelagem de dados e a linguagem SQL.

Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico
Indica nomes de arquivos e referências bibliográficas relacionadas ao longo do texto.

```
Largura constante
```

Indica comandos e suas opções, variáveis e atributos, conteúdo de arquivos e resultado da saída de comandos. Comandos que serão digitados pelo usuário são grifados em negrito e possuem o prefixo do ambiente em uso (no Linux é normalmente # ou $, enquanto no Windows é C:\).

Conteúdo de slide
Indica o conteúdo dos slides referentes ao curso apresentados em sala de aula.

Símbolo
Indica referência complementar disponível em site ou página na internet.

Símbolo
Indica um documento como referência complementar.

Símbolo
Indica um vídeo como referência complementar.

Símbolo
Indica um arquivo de áudio como referência complementar.

Símbolo
Indica um aviso ou precaução a ser considerada.

Símbolo
Indica questionamentos que estimulam a reflexão ou apresenta conteúdo de apoio ao entendimento do tema em questão.

Símbolo
Indica notas e informações complementares como dicas, sugestões de leitura adicional ou mesmo uma observação.

Símbolo
Indica atividade a ser executada no Ambiente Virtual de Aprendizagem – AVA.

Permissões de uso

Comentários e perguntas

Para enviar comentários e perguntas sobre esta publicação:
Escola Superior de Redes RNP
Endereço: Av. Lauro Müller 116 sala 1103 – Botafogo
Rio de Janeiro – RJ – 22290-906
E-mail: info@esr.rnp.br

Sobre o autor

Paulo Henrique Cayres possui graduação no curso Superior de Tecnologia em Processamento de Dados pela Universidade para o Desenvolvimento do Estado e da Região do Pantanal (UNIDERP), especialização em Análise de Sistemas pela Universidade Federal de Mato Grosso do Sul (UFMS) e mestrado em Ciências da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS). Atuou como coordenador curso de Bel. em Sistemas de Informação e Superior de Tecnologia em Redes de Computadores na Faculdade da Indústria do Sistema FIEP, onde também coordenou as atividades do SGI - Setor de Gestão de Informações. Atualmente é coordenador do Núcleo de Educação a Distância - NEaD da Faculdade da Indústria do Sistema FIEP. Sócio-diretor da CPP Consultoria e Assessoria em informática Ltda. Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: linguagens de programação, engenharia de software, modelagem de sistemas, desenvolvimento de aplicações para web e gerência de projetos. Professor titular em cursos de graduação e pós-graduação ministrando disciplinas de desenvolvimento de sistemas desde 1995. Instrutor de treinamento na linguagem Java de programação junto ao CITS em Curitiba e na ESR-RNP.

John Lemos Forman é Mestre em Informática (ênfase em Engenharia de Software) e Engenheiro de Computação pela PUC-Rio, com pós-graduação em Gestão de Empresas pela COPPEAD/UFRJ. É vice-presidente do Sindicato das Empresas de Informática do Rio de Janeiro – TIRIO, membro do Conselho Consultivo e de normas Éticas da Assespro-RJ e Diretor da Riosoft. É sócio e Diretor da J.Forman Consultoria e coordenador acadêmico da área de desenvolvimento de sistemas da Escola Superior de Redes da RNP. Acumula mais de 29 anos de experiência na gestão de empresas e projetos inovadores de base tecnológica, com destaque para o uso das TIC na Educação, mídias digitais e Saúde.

1

Modelagem e uso de Bancos de Dados – Visão geral

objetivos

Aprender sobre bancos de dados e sistemas gerenciadores de bancos de dados;
Entender o conceito de sistema de informação e modelos de sistemas, e etapas da modelagem do banco de dados.

conceitos

Banco de dados; SGBD; SI; Minimundo; Modelos conteitual, lógico e físico.

Exercício de nivelamento
Modelagem de dados

O que você entende por modelagem de dados?

Introdução a Banco de Dados

Banco de dados: coleção de dados relacionados que podem ser inseridos, atualizados, e recuperados e que possuem um significado implícito.

- Dado x informação.
- Exemplos: agenda de celular, catálogo de livros, planilha orçamentária etc.
- Desafios: volume de dados, evitar redundância e inconsistência, acesso, segurança, integridade e migração.

Juntamente com a chegada dos computadores, começaram a ser criados sistemas para atender as necessidades de processamento crescente do volume de dados das empresas. Nessa época, a manipulação de informações era implementada através de módulos isolados que atendiam uma determinada necessidade e que, com o passar do tempo, foram sendo incrementados com novos módulos sobre os já existentes. O problema é que, muitas vezes, os novos módulos eram escritos por outros programadores, que por sua vez não utilizavam as mesmas linguagens. Mais do que isso, os sistemas eram ainda muito primitivos, trabalhando com os sistemas de arquivos disponíveis na época, os quais não controlavam o acesso concorrente por vários usuários ou processos. Com todas essas divergências aconteciam diversos problemas, desde a redundância até o isolamento de dados.

Dados podem ser vistos como uma representação de fatos, conceitos ou instruções de uma maneira normalizada, podendo ser adaptados à comunicação, interpretação e processamento. Já a informação pode ser vista como todo o conjunto de dados devidamente ordenados e organizados de forma significativa. Os bancos de dados, por sua vez, foram concebidos com o objetivo de possibilitar o armazenamento de informações em sistemas de arquivos permanentes, com o intuito de possibilitar posteriores acesso e manipulação de informações, de forma organizada e estruturada.

Os principais desafios a serem alcançados através de um Sistema de banco de dados eram:

- **Gerenciamento de grande quantidade de informação**: um Sistema de banco de dados teria de possibilitar o armazenamento de informações tanto de sistemas simples, como uma agenda telefônica, quanto de sistemas mais complexos, como um sistema de reserva de passagens aéreas. Em ambos os casos o Sistema de banco de dados teria de ser capaz de prover segurança e confiabilidade, independente da quantidade de informações que iria armazenar;
- **Evitar redundância e inconsistência de dados**: um Sistema de banco de dados teria de ter a capacidade de reduzir ao máximo, ou mesmo eliminar, a redundância da informação em lugares diferentes, muito comum nos Sistemas de Arquivos existentes até então. Um dos problemas da redundância é que podemos atualizar um determinado dado de um arquivo e essa atualização não ser feita em todo o sistema – esse problema é chamado de inconsistência;
- **Facilidade de acesso**: um Sistema de banco de dados deveria facilitar ao máximo o acesso aos dados, se preocupando com um possível acesso concorrente, onde podemos ter a mesma informação sendo compartilhada por diversos usuários;
- **Segurança de Dados**: o Sistema de banco de dados deveria garantir a segurança de acesso aos dados por meio da implementação de usuários e senhas de acessos;
- **Garantia de Integridade**: é fazer com que os valores dos dados atribuídos e armazenados em um banco de dados devam satisfazer certas restrições para manutenção de consistência e coerência;
- **Facilidade de Migração**: um Sistema de banco de dados deveria garantir a possível transferência de dados entre Banco de dados. Ao ato de transferir as informações de um banco de dados para outro banco de dados deu-se o nome de Migração.

Histórico.

- Início da computação:
  - Dados guardados em arquivos de texto.
  - Problemas nesse modelo:
    - Redundância e inconsistência não controlada de dados.
    - Aplicações devem se preocupar com a forma de armazenamento dos dados.
  - Início dos anos 60: primeiros SGBDs.

O primeiro Sistema Gerenciador de banco de dados (SGBD) comercial surgiu no final de 1960, levando em consideração todos os desafios propostos para um sistema de banco de dados. Os SGBDs representaram uma evolução considerável em relação aos sistemas de arquivos de armazenamento em disco, criando novas estruturas de dados com o objetivo de armazenar informações.

Sistema Gerenciador de banco de dados

Características de um SGBD:

- Gerenciamento de grande quantidade de dados.
- Evitar redundância e inconsistência de dados.
- Concorrência de acesso.
- Facilidade de acesso.
- Segurança de dados.
- Garantia de integridade.
- Facilidade de migração.
- Suporte a Transações.
- Exemplos: PostgreSQL, MySQL, OracleDB, MS SQLServer etc.

Um Sistema de Gerenciamento de banco de dados (SGBD) é um software que incorpora e facilita as funções de definição, recuperação e alteração de dados em um Banco de Dados. Tem a função de proteção (contra falhas de hardware e software) e de segurança (acessos não autorizados ou maliciosos) dos dados nele armazenados, ao mesmo tempo em que permite o compartilhamento desses dados entre vários usuários e aplicações.

Arquitetura de um SGBD

De forma simplificada, um SGBD faz a interface entre a camada física de armazenamento dos dados (discos, storage, métodos de acesso, clustering de dados etc.) e a sua organização lógica (instâncias) através de um determinado modelo de organização (esquema ou subesquema).

Linguagens de programação e ferramentas front-end visuais gráficas são algumas soluções de software que auxiliam usuários na construção, manutenção e manipulados dos dados armazenados em bancos de dados nos SGBDs. Internamente, um SGBD apresenta linguagens específicas para trabalhar com seus de dados, cuidando da sua definição (Data Definition Language – DDL), manipulação (Data Manipulation Language – DML) e consultas (Query Language).

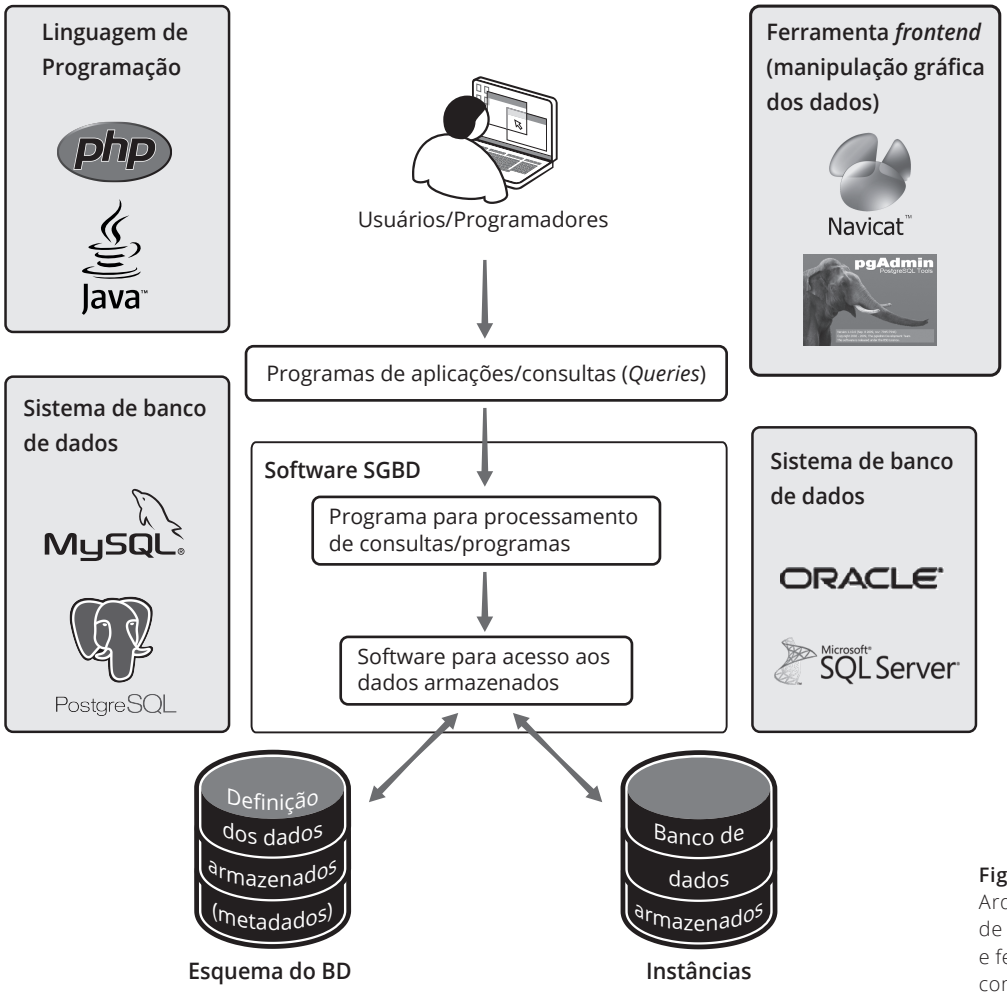


Figura 1.1
Arquitetura de um SGBD e ferramentas complementares.

Organização de SGBDs

Os principais modelos de organização de SGBDs atualmente existentes são:

- Modelo Hierárquico.
- Modelo de Rede.
- Modelo Relacional.
- Modelo Orientado a Objetos.
- Modelo Objeto-Relacional.
- Modelo NoSQL (Not only SQL).

A modelagem de dados representa a descrição formal da estrutura de um SGBD. A evolução tecnológica, tanto de hardware como de software, permitiu que diferentes alternativas para a organização da estrutura de SGBDs fossem testadas e aprimoradas.

Modelo hierárquico

O modelo hierárquico foi o primeiro a ser reconhecido como um modelo de dados. Seu desenvolvimento somente foi possível devido à consolidação dos discos de armazenamento endereçáveis, pois esses discos possibilitaram a exploração de sua estrutura de endereçamento físico para viabilizar a representação hierárquica das informações. Nesse modelo de dados, os dados são estruturados em hierarquias ou árvores. Os nós das hierarquias

Saiba mais

O sistema comercial mais divulgado no modelo hierárquico foi o Information Management System da IBM Corp (IMS).

contêm ocorrências de registros, onde cada registro é uma coleção de campos (atributos), cada um contendo apenas uma informação. O registro da hierarquia que precede a outros é o registro-pai, os outros são chamados de registros-filhos.

Grande parte das restrições e consistências de dados estava contida dentro dos programas escritos para as aplicações. Era necessário escrever programas na ordem para acessar o banco de dados.

Modelo em rede

O modelo em redes surgiu como uma extensão ao modelo hierárquico, eliminando o conceito de hierarquia e permitindo que um mesmo registro estivesse envolvido em várias associações. No modelo em rede, os registros são organizados em grafos, onde aparece um único tipo de associação (set) que define uma relação 1:N entre 2 tipos de registros: proprietário e membro.

O gerenciador Data Base Task Group (DBTG), da Committee on Data Systems and Languages (CODASYL) estabeleceu uma norma para esse modelo de banco de dados, com linguagem própria para definição e manipulação de dados.

Os dados tinham uma forma limitada de independência física. A única garantia era a de que o sistema deveria recuperar os dados para as aplicações como se eles estivessem armazenados na maneira indicada nos esquemas. Os geradores de relatórios da CODASYL também definiram sintaxes para dois aspectos chaves dos sistemas gerenciadores de dados: concorrência e segurança. O mecanismo de segurança fornecia uma facilidade na qual parte do banco de dados (ou área) pudesse ser bloqueada para prevenir acessos simultâneos, quando necessário. A sintaxe da segurança permitia que uma senha fosse associada a cada objeto descrito no esquema.

Saiba mais

No Modelo em Rede, o sistema comercial mais divulgado é o CA-IDMS da Computer Associates.

Ao contrário do Modelo Hierárquico, em que qualquer acesso aos dados passa pela raiz, o modelo em rede possibilita acesso a qualquer nó da rede sem passar pela raiz.

Modelo relacional

O modelo relacional apareceu devido às seguintes necessidades:

- Aumentar a independência de dados nos sistemas gerenciadores de banco de dados;
- Prover um conjunto de funções apoiadas em álgebra relacional para armazenamento e recuperação de dados;
- Permitir processamento dedicado e exclusivo.

O modelo relacional, tendo por base a teoria dos conjuntos e álgebra relacional, foi resultado de um estudo teórico realizado por um pesquisador da IBM chamado Ted Codd, que escreveu um artigo na década de 70 propondo um novo modelo para armazenamento e recuperação de dados.

O modelo relacional revelou-se ser o mais flexível e adequado ao solucionar os vários problemas que se colocam no nível da concepção e implementação das bases de dados. A estrutura fundamental do modelo relacional é a relação (tabela). Uma relação é constituída por um ou mais atributos (campos) que traduzem o tipo de dados a armazenar. Cada instância do esquema (linha) é chamada de tupla (registro).

O modelo relacional não tem caminhos predefinidos para se fazer acesso aos dados como nos modelos que o precederam. O modelo relacional implementa estruturas de dados organizadas em relações. Porém, para trabalhar com essas tabelas, algumas restrições

precisaram ser impostas para evitar aspectos indesejáveis, como: repetição de informação, incapacidade de representar parte da informação e perda de informação. Essas restrições são: integridade referencial, chaves e integridade de junções de relações.

Banco de dados relacional:

- Item de dado: campo, coluna, atributo;
- Registro: linha, tupla;
- Tabela;
- Manipulação com SQL: operações CRUD.

Modelo Orientado a Objetos

Os bancos de dados Orientados a Objeto começaram a se tornar comercialmente viáveis em meados de 1980. A motivação para seu surgimento está em função dos limites de armazenamento e representação semântica impostas no modelo relacional. Alguns exemplos são os sistemas de informações geográficas (SIG), os sistemas CAD e CAM, que são mais facilmente construídos usando tipos complexos de dados. A habilidade para criar os tipos de dados necessários é uma característica das linguagens de programação orientadas a objetos.

Saiba mais

O termo Modelo Orientado a Objetos é usado para documentar o padrão que contém a descrição geral das facilidades de um conjunto de linguagens de programação orientadas a objetos e a biblioteca de classes que pode formar a base para o sistema de banco de dados.

Contudo, esses sistemas necessitam guardar representações das estruturas de dados que utilizam no armazenamento permanente. A estrutura padrão para os bancos de dados orientados a objeto foi feita pelo Object Database Management Group (ODMG). Esse grupo é formado por representantes dos principais fabricantes de banco de dados orientados a objeto disponíveis comercialmente. Membros do grupo têm o compromisso de incorporar o padrão em seus produtos.

Quando os bancos de dados orientados a objetos foram introduzidos, algumas das falhas perceptíveis do modelo relacional pareceram ter sido solucionadas com essa tecnologia e acreditava-se que tais bancos de dados ganhariam grande parcela do mercado. Hoje, porém, acredita-se que os bancos de dados Orientados a Objetos serão usados em aplicações especializadas, enquanto os sistemas relacionais continuarão a sustentar os negócios tradicionais, onde as estruturas de dados baseadas em relações são suficientes. O diagrama de classes UML serve geralmente como o esquema para o modelo de dados Orientado a Objetos.

Modelo Objeto-Relacional

Alguns bancos de dados relacionais adicionaram a seus produtos a capacidade de incorporar objetos mais complexos, como imagem, som e vídeo, bem como alguns recursos de orientação a objetos.

No entanto, isso não os torna sistemas puramente orientados a objetos, apesar da denominação Object-Relational Database Management System (ORDMS). Esse modelo prevê a implementação de uma camada de abstração de dados em cima dos métodos relacionais, o que torna possível a manipulação de dados mais complexos.

Modelo NoSQL

Um dos grandes desafios atualmente na área de computação é a manipulação e processamento de grande quantidade de dados no contexto de Big Data.

NoSQL (Not only SQL: Não só SQL) é um termo utilizado para definir um tipo de banco de dados que não segue normas de tabelas (schemas) presente no banco de dados relacional.

A quantidade de dados gerada diariamente em vários domínios de aplicação como, por exemplo, da web, rede sociais, redes de sensores, dados de sensoriamento, entre diversos outros, estão na ordem de algumas dezenas, ou centenas, de Terabytes.

Uma das tendências para solucionar os diversos problemas e desafios gerados pelo contexto Big Data é o movimento denominado NoSQL. NoSQL promove diversas soluções inovadoras de armazenamento e processamento de grande volume de dados. Essas soluções foram inicialmente criadas para solucionar problemas gerados por aplicações, por exemplo, web 2.0, que na sua maioria necessitam operar com grande volume de dados, tenham uma arquitetura que "escale" com grande facilidade de forma horizontal, permitam fornecer mecanismos de inserção de novos dados de forma incremental e eficiente, além da necessidade de persistência dos dados em aplicações nas nuvens (cloud computing).

Sistema de Informação

Um Sistema de Informação (SI) é um conjunto de elementos ou componentes inter-relacionados que coleta (entrada), manipula (processo), armazena e dissemina dados (saída) e informações, além de fornecer um mecanismo de realimentação (ação corretiva) para garantir a realização de um determinado objetivo.

Informações são resultados obtidos pela seleção, sumarização e apresentação de dados de uma forma que seja útil aos interessados. Geralmente é obtida quando alguma atividade mental humana (observação e análise) é realizada com sucesso sobre dados para revelar seu significado ou sentido. Assim, vale novamente tecer algumas considerações sobre Dados x Informação.

Dados x Informação:

- **Dados**: fatos que podem ser armazenados. Exemplo: cor dos olhos, idade, sexo etc.
- **Informação**: realidade observada sobre os dados. Exemplo: quantidade de pessoas com cor dos olhos castanho, a média de idade das pessoas do sexo masculino.
  - **Importância da Informação**: necessidade de qualidade, eficácia, informações mais confiáveis e rápidas para a tomada de decisão.
  - **Uso de recursos da TI**: envolve hardware, software, pessoas, banco de dados, redes, procedimentos

Os dados são constituídos de fatos crus, geralmente representando dados do mundo real, como o número de um funcionário ou total de horas trabalhadas, podendo ser de vários tipos: numéricos, alfanuméricos, imagem, vídeo, som etc. O processo de organização desses fatos crus de forma significativa os tornam uma informação.

Informação é um conjunto de fatos organizados de tal maneira que possuem valor adicional, além do valor dos fatos individuais (crus). Por exemplo, os gerentes de vendas de uma determinada empresa podem crer que conhecer o total de vendas de sua empresa em um determinado período seja mais importante do que o número de vendas de cada um de seus funcionários. Já o funcionário deve se preocupar com o seu total de vendas frente a uma possível meta a ser alcançada.

Transformar dados em informação é um processo, ou um conjunto de tarefas logicamente relacionadas realizada para alcançar um resultado definido.

Atualmente esse conjunto de tarefas envolve um conjunto de pessoas, procedimentos, softwares, hardwares, bancos de dados e mecanismos para criar, armazenar e usar o conhecimento e a experiências existentes nas organizações.

Desenvolvimento de SI

O desenvolvimento de sistemas envolve criar ou modificar os sistemas de negócio existentes. As principais etapas desse processo e seus objetos incluem investigação de sistemas (ter entendimento claro de qual é o problema), análise de sistemas (definir o que o sistema deve fazer para resolver o problema), projeto de sistema (determinar exatamente como o sistema vai funcionar para atender as necessidades do negócio), implantação do sistema (criar ou adquirir os vários componentes do sistema definidos nas etapas de projeto) e manutenção e revisão do sistema (manter e depois modificar o sistema para que ele continue a atender as necessidades evolutivas do negócio).

Para que essas etapas possam ser alcançadas, podemos partir de um plano de que propõe um ciclo de vida com vistas à organização para o processo de desenvolvimento de um SI, quais sejam:

Desenvolvimento de um SI: informalmente, é iniciado como resultado de um "entendimento verbal" entre o contratante e o desenvolvedor.

- Plano de desenvolvimento (organização): CICLO DE VIDA.
  - Definir atividades a serem executadas;
  - Verificar consistência entre as atividades;
  - Introduzir pontos de verificação e validação.

Nesse ponto temos de ter em mente que há uma diferença entre Sistemas de Informações e o software propriamente dito. O primeiro é formado de partes que interagem entre si, visando um objetivo comum, tais como software, hardware e recursos humanos. O segundo é um conjunto de instruções de programas desenvolvidas para resolver um problema com o uso da computação.

Outra necessidade importante é a de entendermos o papel do DBA e do Analista de Dados no processo de desenvolvimento de um SI. DBA (DataBase Administrator) é o profissional responsável por gerenciar, instalar, configurar, atualizar e monitorar um banco de dados ou sistemas de bancos de dados, ao passo que o Analista de Dados é o profissional responsável por atuar com administração de banco de dados, desenvolver melhorias, identificar e solucionar problemas.

Desenvolvimento de software x SGBD

Um SGBD, que é um tipo especial de software, é um componente importante de um sistema ou aplicação que fará uso dos dados nele armazenados. O SGBD em si será um componente que não precisará ele próprio ser desenvolvido, mas a organização de como os dados serão criados e armazenados dependerá diretamente dos objetivos pretendidos para o sistema ou aplicação que será desenvolvido.

Existem várias metodologias (modelos gerais, paradigmas) de desenvolvimento de software, mas em geral poderemos sempre identificar as seguintes fases:

- Especificação (dos requisitos);
- Análise e Projeto (construção dos modelos);
- Implementação (desenvolvimento/codificação);
- Testes/Homologação (implantação);
- Manutenção.

Ainda que faça parte do escopo desse curso descrever em detalhes diferentes metodologias de desenvolvimento de software, é importante relembrar essas etapas, com destaque para a primeira etapa.

As especificações de requisitos são objetivos ou restrições estabelecidas por clientes e usuários do sistema que definem suas diversas propriedades. Os requisitos de software são, obviamente, aqueles entre os requisitos de sistema que dizem respeito a propriedades do software. Dessa forma a especificação de requisitos envolve as atividades de determinar os objetivos de um software e as restrições associadas a ele. Ela deve também estabelecer o relacionamento entre esses objetivos e restrições e a especificação precisa do software.

Tradicionalmente, os requisitos de software são separados em requisitos funcionais, com a descrição das diversas funções que clientes e usuários querem ou precisam que o software ofereça, e não funcionais, com as qualidades globais de um software, como manutenibilidade, usabilidade, desempenho, custos e várias outras.

Entrevista, observação in-loco e encontros são vistos como algumas das técnicas que podem ser utilizadas na fase de especificação de requisitos.

As demais fases de análise e projeto, implementação, teste e manutenção, que complementam um ciclo tradicional de desenvolvimento de software, buscam traduzir o conjunto de requisitos levantados em uma implementação que permita atingir os objetivos propostos (na etapa de manutenção o ciclo se repete de modo a permitir que seja incorporadas ao sistema possíveis melhorias que se fizerem necessárias).

Nosso interesse está com foco na etapa inicial, já que a modelagem do banco de dados dependerá diretamente do levantamento dos requisitos que o futuro sistema terá de atender.

Modelos de Sistema

Tipos de Modelos:

- Funcional (processos).
- De Dados.
- Comportamental (tempo-dependente).

Antes de construir um SI, deve-se elaborar um modelo (planta) que seja capaz de expressar, com a máxima fidelidade e simplicidade possíveis, o ambiente no qual ele se insere visando satisfazer todos os requisitos identificados. Isso nos possibilita ter um maior domínio sobre o problema, uma vez que o modelo facilita e diminui os custos de possíveis adequações do projeto.

Cada um dos modelos recém citados busca retratar diferentes aspectos ou dimensões do sistema a ser desenvolvido, sendo vistos como um mecanismo eficaz de comunicação entre técnicos e usuários. Esses modelos lidam também com a questão da complexidade através de níveis de abstração, tornando mais fáceis (e também mais baratas) as discussões em torno da adequação do sistema que será construído. A seguir detalhamos um pouco mais cada um desses modelos.

Funcional

Pode ser visto como sendo formado por uma representação gráfica, acompanhada de uma descrição de cada função e das suas interfaces. Costuma ser expressa por meio de Diagrama de Fluxo de Dados (DFD).

De dados

Método da análise de sistemas que busca especificar, a partir de fatos relevantes que estejam associados ao domínio de conhecimento analisado, a perspectiva dos dados, permitindo organizá-los em estruturas bem definidas, estabelecendo as regras de dependência entre eles, produzindo um modelo expresso por uma representação, ao mesmo tempo descritiva e diagramática.

Comportamental

Descreve de que maneira o sistema, enquanto um conjunto de elementos inter-relacionados, reage, internamente, com um todo organizado, aos estímulos externos. Preocupa-se em mostrar as ações que o sistema deve executar para responder adequadamente aos eventos previstos no modelo ambiental.

Projeto de banco de dados

Três níveis de abstração (necessário realizar o mapeamento entre os três modelos):

- Modelo Conceitual (DER).
- Modelo Lógico (Esquema do BD).
- Modelo Físico (Script do BD em SQL).

O projeto de concepção e implementação do banco de dados que vai dar suporte ao sistema sendo desenvolvido deve, igualmente, passar por diferentes estágios de abstração. São enormes as pressões para que a equipe de informática possa responder o mais rapidamente possível às demandas para colocar em funcionamento o novo sistema. Assim, não é incomum tomar conhecimento de projetos de desenvolvimento onde as etapas de levantamento de requisitos e modelagem são relegadas a um segundo plano. Esse é um erro comum que costuma cobrar seu preço, em geral muito significativo, nas etapas seguintes do processo de desenvolvimento. Nossa recomendação é não negligenciar nenhuma das seguintes etapas ao projetar um banco de dados.

- **Modelo Conceitual (DER)**: também conhecido como Diagrama Entidade-Relacionamento, é um modelo de dados abstrato que descreve a estrutura de um banco de dados independe de sua implementação;

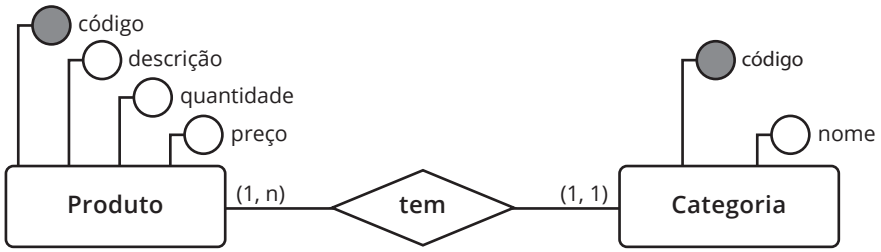


Figura 1.2
Exemplo de um Modelo Conceitual.

- **Modelo Lógico (Esquema do BD)**: tem como objetivo transformar o modelo conceitual em um modelo que define como o banco de dados será implementado em um SGBD específico. Deve representar relações e restrições do modelo de dados que representa a estrutura de um BD e o Esquema do Banco de Dados;

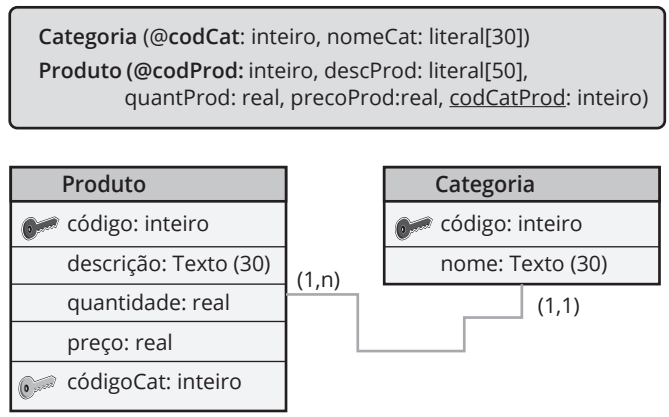


Figura 1.3
Exemplo de um Modelo Lógico.

- **Modelo Físico (Script do BD em SQL)**: nessa fase, o modelo do banco de dados é enriquecido com detalhes que influenciam no desempenho do banco de dados, mas não interferem na sua funcionalidade. Script do banco de dados em SQL representa os detalhes dos dados internamente ao BD (campo, tipo/domínio, restrições).

```
CREATE TABLE PRODUTO (
   codprod INTEGER PRIMARY KEY,
   quantprod REAL,
   precoprod REAL,
   descrprod VARCHAR(30),
   codcatprod INTEGER
);
CREATE TABLE CATEGORIA (
   codcat INTEGER PRIMARY KEY,
   nomecat VARCHAR(30)
);
ALTER TABLE PRODUTO ADD FOREIGN KEY(codcatprod) REFERENCES
CATEGORIA (codcat);
```

Etapas da modelagem do banco de dados

A figura a seguir ilustra o esquema geral das diferentes etapas que serão percorridas ao longo do desenvolvimento de um novo banco de dados no contexto do desenvolvimento de um novo sistema ou aplicação.

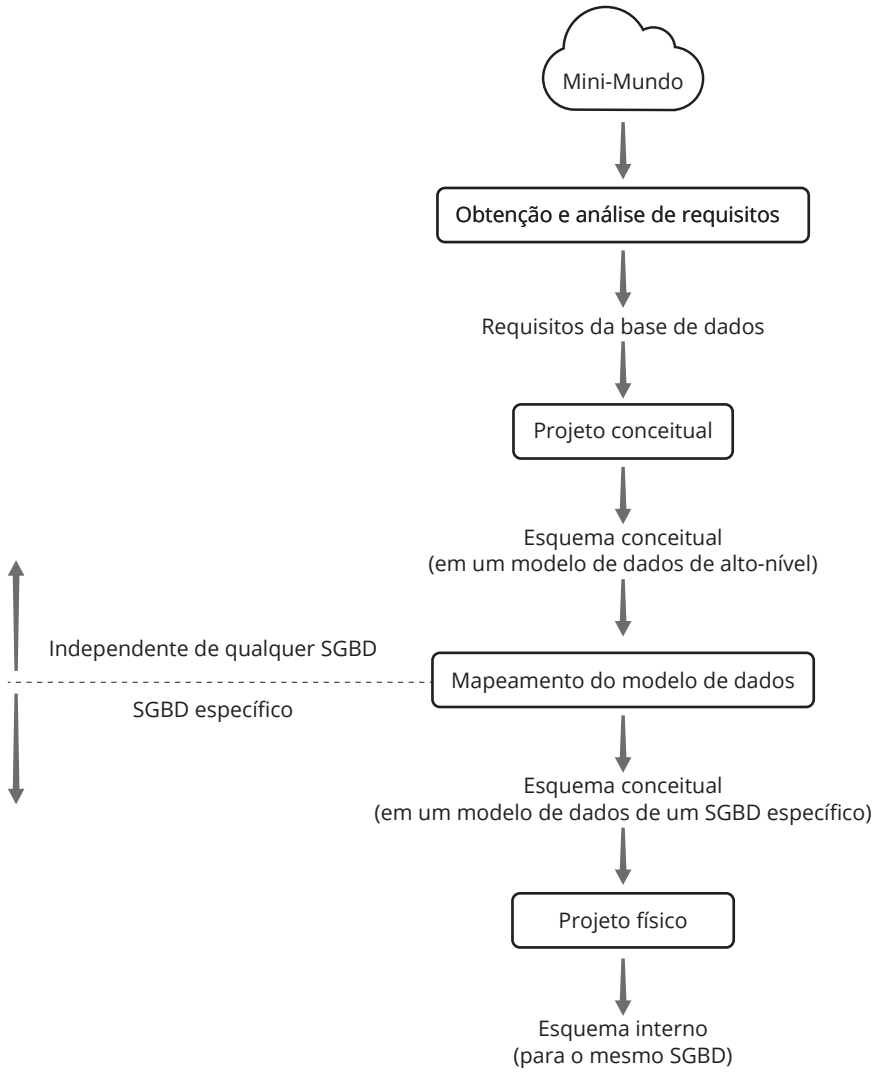


Figura 1.4
Esquema geral de modelagem de dados com o uso de ER.

Primeira Etapa – Análise de Requisitos

Levantamento/Estudo de Viabilidade:

- Identificar usuários responsáveis e definir escopo.
- Identificar as deficiências no sistema atual (justificativa).
- Estabelecer metas e objetivos para o novo sistema.
- Determinar se é possível "informatizar" (viabilidade).
- Fazer estimativas (cálculo de Custo/Benefício).
- Preparar um cronograma.

É indispensável que o profissional ou equipe envolvidos com o projeto do banco de dados tenha um bom domínio do negócio da sua organização ou empresa. Para tanto, pode e deve fazer uso de diferentes ferramentas para a coleta de dados, tais como entrevistas, análise de procedimentos e documentos, questionário etc. O objetivo é identificar requisitos de dados (fatos do mundo real) que deverão ser observados ou atendidos.

Essa é uma etapa pré-modelagem, onde o mais comum é elaborar a especificação formal dos dados no formato de um texto descritivo, também chamado de minimundo ou Universo de Discurso, que pode ser validado pelos futuros usuários do sistema em desenvolvimento. A seguir, apresentamos um exemplo de minimundo.

Exemplo de Minimundo: EMPRESA

A **empresa** está organizada em departamentos. Cada **departamento** tem um nome, um número único e um empregado que gerencia o departamento. Armazena-se a data em que o empregado começou a gerenciar o departamento. Um departamento pode ter diversas localizações;

Um departamento controla inúmeros **projetos**, sendo que cada um tem um nome, um número único e uma localização;

Do **empregado** armazena-se o nome, o número do seguro social, endereço, salário, sexo e data de nascimento. Todo empregado é alocado em um departamento, mas pode trabalhar em diversos projetos, que não são necessariamente controlados pelo mesmo departamento. Armazena-se, também, o número de horas semanais que o empregado trabalha em cada projeto. Mantém-se, ainda, a indicação do supervisor direto de cada empregado;

É feito um controle sobre os dependentes de cada empregado para fins de seguro. De cada **dependente** é registrado o nome, sexo, data de nascimento e o parentesco com o empregado.

FONTE: ELMASRI, R.; NAVATHE, S. B. Sistemas de banco de dados. 4. ed. São Paulo: Pearson, 2005.

Segunda Etapa – Análise e projeto

- Projeto do BD.
- 1) Modelo Conceitual (DER).
- 2) Modelo Lógico (Esquema do BD/Relações).
- 3) Modelo Físico (Script do BD/Tabelas-Restrições).

Com base no texto descritivo, ou minimundo, gerado na primeira etapa, inicia-se o processo de modelagem propriamente dito, indo do mais abstrato para o mais concreto. Assim, os modelos vão se sucedendo conforme indicado na figura 1.4. Nas próximas sessões, cada um desses modelos serão tratados separadamente.

Atividade de Fixação

Descreva com suas palavras. Qual é o principal objetivo de se utilizar o processo de modelagem de dados no processo de desenvolvimento de sistemas?

2

Modelo conceitual: DER

objetivos

Conhecer os conceitos de Diagrama Entidade Relacionamento (DER) e o processo de construção de um DER, fazendo uso de ferramenta CASE e técnicas de normalização.

conceitos

DER; EER; Entidades; Atributos; Relacionamentos; CASE; Dado temporal e normalização.

Diagrama Entidade Relacionamento: DER

De forma resumida, o DER tem as seguintes características:

- Também conhecido como Modelo E-R.
- Definido por Peter Chen (1976), com base na teoria relacional criada por E. F. Codd (1970).
- Estudiosos (Theorey, Fry, James Martin e outros) evoluíram e expandiram o "meta-modelo" (visão moderna) > Engenharia da Informação.
- Objetivo: apresentar uma visão única, não redundante e resumida, dos dados de uma aplicação.

O Diagrama Entidade-Relacionamento (DER) é um modelo conceitual de alto nível, criado na década de 70, e que é empregado no desenvolvimento de projetos de aplicações que vão manipular Banco de Dados. Seu objetivo é o de facilitar a compreensão por parte do usuário, sendo visto como uma ferramenta útil durante o processo de projeto da base de dados, descartando detalhes de como os dados serão armazenados.

A figura 2.1 apresenta uma solução proposta de modelagem de alto nível utilizando o DER para o estudo de caso EMPRESA apresentado na sessão de aprendizagem 1.

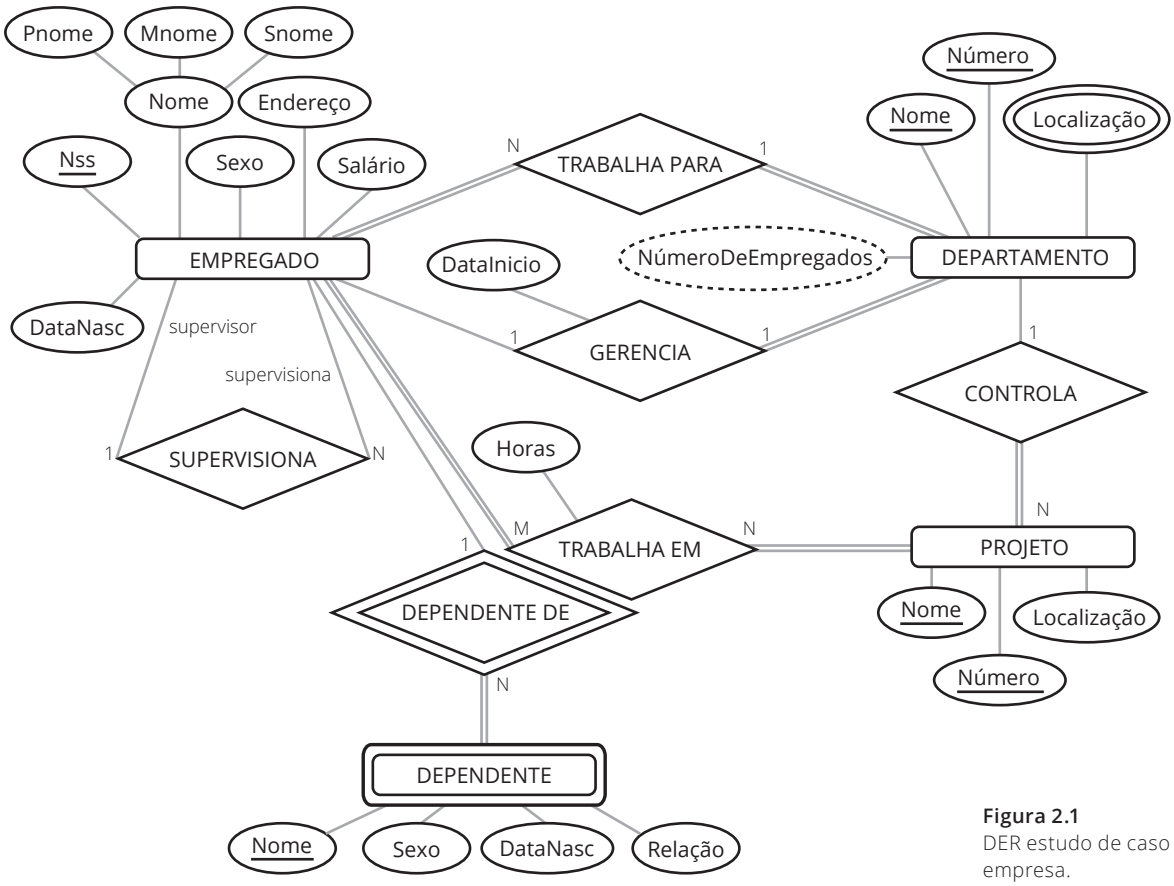


Figura 2.1
DER estudo de caso empresa.

Componentes do DER

Um DER é composto pelos seguintes elementos:

- Entidades: objeto do mundo real com identificação distinta e com um significado próprio.
- Atributos: qualificadores de uma entidade (características que a descrevem).
- Relacionamentos: dependência entre entidades associadas: quando um atributo de uma entidade refere-se a outra.
  - Restrições em relacionamentos: limitam a possibilidade de combinações de entidades que podem participar do relacionamento (restrições estruturais).

A seguir, analisamos cada um deles com detalhes.

Entidades

Uma entidade tem as seguintes características:

- Representa uma classe de dados. Suas instâncias (ocorrências) são a representação desses dados.
- Representação: retângulo com nome em seu interior, sendo que o nome deve estar no singular, representando o conjunto (de instâncias).
- Possui atributos: qualificadores de uma entidade (características que a descrevem).
  - Notação original (Chen, 1976): elipses.

Uma entidade por ser vista como um conjunto de objetos do mundo real que está sendo modelado e sobre o qual desejamos manter informações em um banco de dados.

Em um DER, uma entidade é representada por meio de um retângulo que contém o nome da entidade que se deseja modelar, conforme ilustrado na figura 2.2.

Figura 2.2
Representação gráfica de entidade no DER.

Uma Entidade pode ser um objeto (livro), uma pessoa (empregado), abstrato (curso), acontecimento (inscrição). O nome depende do contexto (pessoa: Aluno, Professor, Segurado, Contribuinte, Empregado).

Atributos

O atributo corresponde a uma dado que é associado a cada ocorrência de uma entidade ou relacionamento. Atributos são representados graficamente conforme ilustrado na figura 2.3.

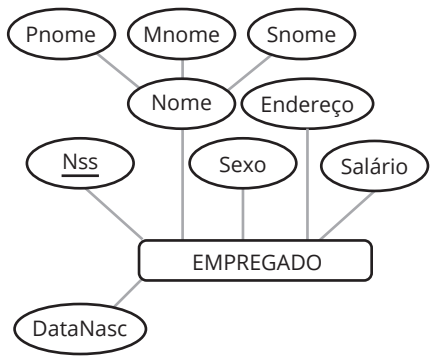


Figura 2.3
Representação de atributos da entidade Empregado.

Na prática, muitas vezes os atributos não são representados graficamente para não sobrecarregar os diagramas, já que entidades podem possuir um grande número de atributos. Nesses casos é preferível o uso de representação textual.

É importante ressaltar que toda entidade deve ter pelo menos um atributo, sendo que deve apresentar atributo identificador:

- Valor sempre distinto para cada instância, caracterizando que não existem objetos repetidos;
  - Restrição de unicidade ou chave primária.
- Não pode ser um valor nulo (vazio, desconhecido);
- Notação original (Chen, 1976): nome sublinhado na elipse.

Atributos podem ser:

Característica	Alternativa
identificador DEPARTAMENTO: SiglaDepto (Reconhece uma única instância na entidade)	não-identificador DEPARTAMENTO: NomeDepto (qualquer outro atributo)
simples EMPREGADO: Cidade	composto EMPREGADO: Endereço (formado de rua, número, bairro, cep, etc.)
monovalorado ALUNO: Matrícula	multivalorado ALUNO: Telefone (pode ser que tenha mais de um: res, com, cel)
derivado NOTA FISCAL: ValorTotal (obtido a partir de outros atributos)	não-derivado NOTA FISCAL: Quantidade (qualquer outro atributo)
relacionante DEPARTAMENTO E EMPREGADO: SiglaDepto (elo de relacionamento entre 2 entidades)	descritivo DEPARTAMENTO: NomeDepto (qualquer outro atributo)
único EMPREGADO: NúmeroMatrícula	não-único EMPREGADO: Endereço (empregados que moram na mesma residência)
obrigatório EMPREGADO: Nome	opcional EMPREGADO: NomeCônjuge (empregado que não é casado)

PRIMARY KEY — Nome sublinhado na elipse

Elipses a partir de elipse

Elipse com linha dupla

TRIGGER — Elipses com linha tracejada

FOREIGN KEY — Só aparece na entidade de origem

UNIQUE

NOT NULL

Figura 2.4
Tipos de atributos.

Atributos Identificadores

Conforme já mencionado, toda entidade tem sempre pelo menos um atributo. Quando o atributo permite distinguir uma ocorrência das demais ocorrências de uma mesma entidade, ele é considerado um especial conhecido como atributo identificador de entidade. Note, contudo, que um atributo identificador corresponde a um conjunto de um ou mais atributos ou relacionamentos. A figura 2.5 apresenta um atributo concatenado, que é um atributo identificador composto por mais do que um atributo, nesse caso composto pelos atributos Nome e Número.

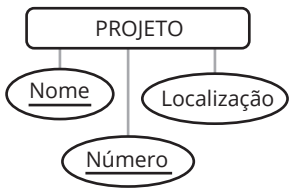


Figura 2.5
Atributo identificador concatenado.

Outro aspecto a ser observado é a característica do atributo identificador, que pode ser inerente à entidade ou dependente de uma entidade ou relacionamento externo. No primeiro caso, a entidade que possui um atributo identificador próprio é chamada de entidade primária ou entidade forte. Já quando a entidade não pode ser identificada somente através de seus próprios atributos, muitas vezes dependendo de um relacionamento para poder ser identificada, ela é chamada de entidade fraca, sendo representada por um retângulo com linha dupla conforme demonstrado na figura 2.6. A seguir apresentamos alguns exemplos para ajudar a compreender esses conceitos.

- Entidade primária ou entidade forte:
  - Atributo próprio (CPF, número da placa);
  - Código atribuído pelo sistema (Número de Matrícula, Código de Fornecedor).
- Entidade fraca:
  - Um dependente, que precisa do nome do seu pai ou mãe.
  - Um item em uma Nota Fiscal, que precisa do número da Nota Fiscal e possivelmente o código do produto correspondente.

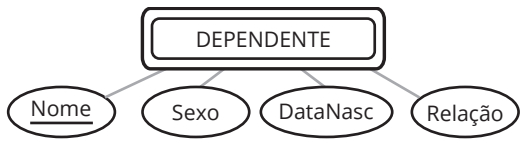


Figura 2.6
Entidade Fraca.

Relacionamentos

Um Relacionamento é a representação de um conjunto de associações entre as ocorrências de entidades. Dessa forma, podemos representar as interações existentes no mundo real, que foram identificadas no processo de análise, entre as entidades. Em um DER, um relacionamento é representado por uma linha reta ligando as entidades relacionadas e, em geral, tem como nome um verbo que represente o contexto da relação em questão. A figura 2.7 apresenta alguns exemplos de relacionamentos utilizando a notação original, onde o nome do relacionamento aparece dentro de um losango na interligação de suas respectivas entidades.

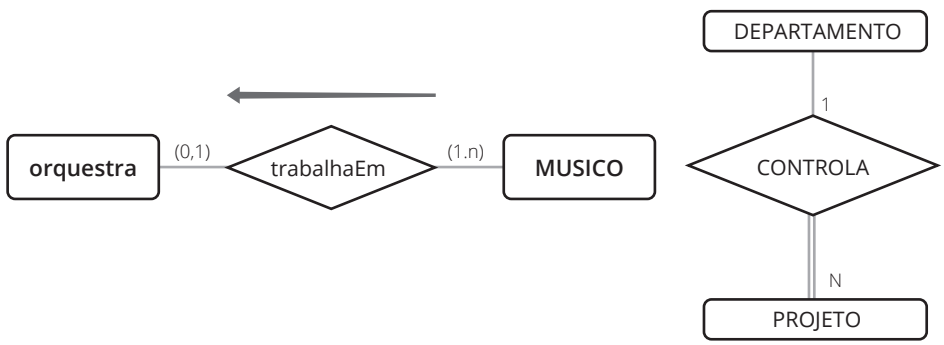


Figura 2.7
Exemplos de relacionamentos (notação original).

Vamos imaginar uma situação onde, após identificar requisitos de dados (fatos do mundo real) tenhamos a seguinte situação:

```
- Pessoas moram em Apartamentos
- Apartamentos formam Condomínios
- Condomínios localizam-se em Ruas ou Avenidas
- Ruas ou Avenidas estão em uma Cidade
```

Notem que os verbos sublinhados nos orientam no processo de diagramação dos relacionamentos entre as entidades destacadas em itálico. Dessa forma, podemos propor o DER apresentado na figura 2.8.

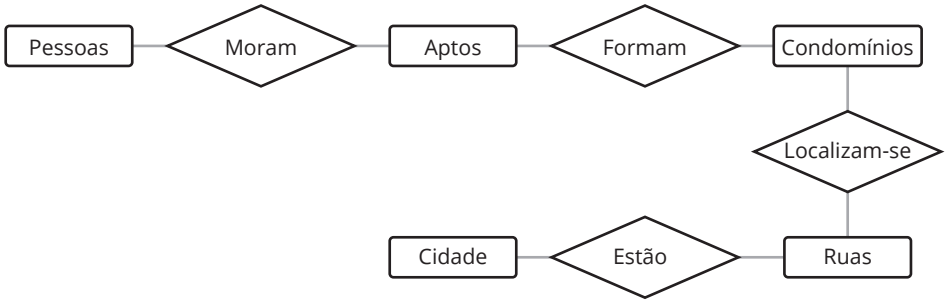


Figura 2.8
Processo de identificação de Relacionamentos entre entidades.

Restrições em relacionamento

No processo de modelagem é importante identificar não apenas as entidades e seus relacionamentos, mas também estabelecer a quantidade de ocorrências de cada um desses relacionamentos. A essa propriedade damos o nome de cardinalidade, que se desdobra em cardinalidade máxima e a cardinalidade mínima.

Em qualquer relacionamento, deve-se questionar se existem restrições estruturais, que são aquelas que limitam a sua cardinalidade. Tomando como exemplo o relacionamento entre Mulher e Filho, há que se perguntar se toda mulher terá necessariamente um filho ou se, tendo filhos, existe um limite superior de filhos que podem ser associados a uma única Mulher. Assim, a cardinalidade poderá revelar diferentes características:

- **Máxima**: razão de Cardinalidade.
- **Mínima**: participação ou dependência de Existência.

A cardinalidade máxima, também chamada de Razão de Cardinalidade/Conectividade, informa o número de ocorrências (máximo) de instâncias de uma entidade em outra. É representada, na notação original, através dos símbolos 1 ou N de cada lado dos losangos do relacionamento.

Existem 3 variações possíveis:

- **1:1** – cada instância de uma entidade relaciona-se com uma e somente uma instância da outra:
  - Motorista possui CNH, Homem casado com Mulher.
- **1:N** (ou N:1) – uma instância relaciona-se com várias na outra entidade, mas cada instância da outra entidade só pode estar relacionada a uma única ocorrência da primeira entidade:
  - Venda feita para Cliente, Filme tem Gênero.
- **N:N** (ou N:M) – uma instância relaciona-se com várias ocorrências na outra entidade, e vice-versa:
  - Estudante cursa Disciplina, Venda contém Produto.

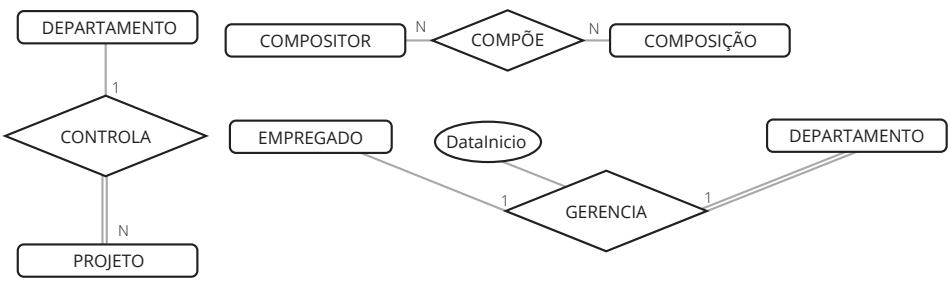


Figura 2.9
Exemplos de restrições de relacionamento.

Diagrama de ocorrências

O diagrama de ocorrências é uma ferramenta utilizada na identificação de quantidades em possíveis relações entre entidades durante o processo de análise.

No relacionamento de grau 1:1 (Um-para-Um), cada elemento de uma entidade relaciona-se com um e somente um elemento de outra entidade. Já no relacionamento de grau 1:N (Um-para-Muitos), cada elemento da entidade 1 relaciona-se com muitos elementos da entidade 2. Esse grau de relacionamento é o mais comum no mundo real, sendo visto como o relacionamento básico entre entidades. Por fim, temos o relacionamento de grau M:N (Muitos-para-Muitos), onde cada elemento da entidade 1 relaciona-se com muitos elementos da entidade 2 e cada elemento da entidade 2 relaciona-se com muitos elementos da entidade 1. Isso ocorre uma vez que em ambos os sentidos de leitura do relacionamento encontra-se um grau Um-para-Muitos, o que caracteriza um contexto geral de Muitos-para-Muitos.

Na figura 2.10 a seguir, podemos ver exemplos do diagrama de ocorrências em relacionamentos de diferentes cardinalidades

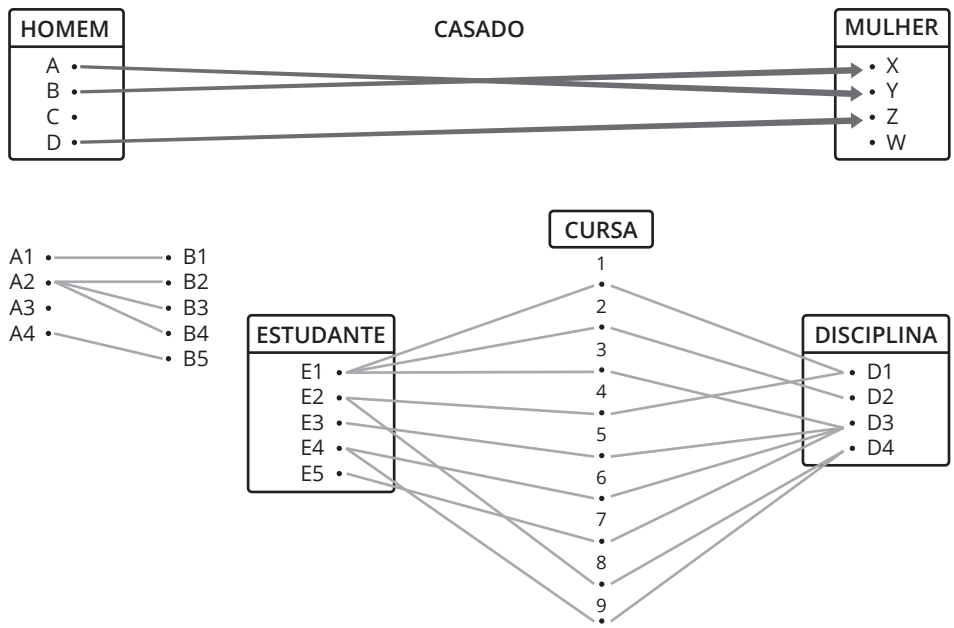


Figura 2.10
Diagramas de ocorrências..

Voltando ao conceito de Cardinalidade Mínima, já apresentado antes, é importante destacar que essa característica indicará a Participação ou Dependência de Existência entre diferentes entidades.

Para a cardinalidade mínima, temos duas possibilidades:

- Total/Obrigatória (dependência de existência): Empregado trabalha para Departamento:
  - Representação por linha dupla no relacionamento.
- Parcial/Opcional: Empregado gerencia Departamento.

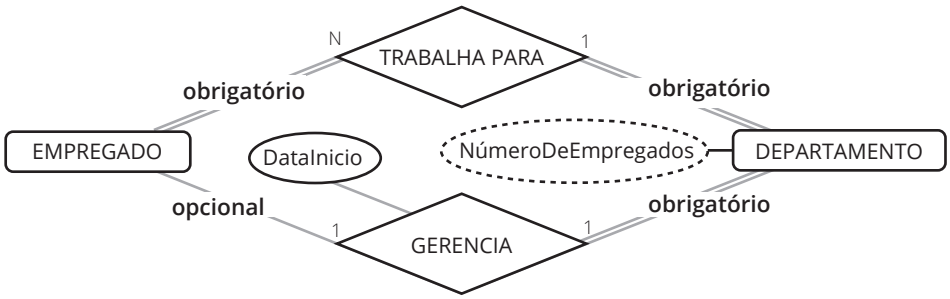


Figura 2.11
Exemplos de participação/ dependência em relacionamentos.

CrowsFoot

Nos exemplos apresentados até o momento, temos privilegiado o uso da notação original empregada quando o DER começou a ser difundido. Com o passar do tempo, outras notações foram sendo sugeridas, sempre buscando aprimorar o processo de análise/confecção dos Diagramas. Uma notação bastante utilizada e que representa a cardinalidade de relacionamentos de forma diferente é a notação CrowsFoot. Na figura a seguir, apresentamos uma tabela com essa notação.

Extremidade	Mín.	Máx.	Descrição
A — B	1	1	Cada instância de A está associada a uma única instância de B
A — B	1	N	Cada instância de A está associada a uma ou várias instâncias de B
A — B	0	1	Cada instância de A está associada a zero ou uma única instância de B
A — B	0	N	Cada instância de A está associada a zero, uma ou várias instâncias de B

Figura 2.12
Exemplo de modelagem com a notação CrowsFoot.

A notação CrowsFoot foi proposta por James Martin sob a ótica de engenharia de informação, sendo também conhecida como "pé-de-galinha", onde os relacionamentos são representados apenas por uma linha.

Nesta notação, podemos representar apenas relacionamentos binários e a representação de cardinalidade máxima e mínima são gráficas. Sendo assim, o símbolo mais próximo da entidade (retângulo) é a representação da cardinalidade máxima e o mais distante da cardinalidade mínima.

Grau do Relacionamento

- Unário (grau 1): relacionamento com a própria entidade, também chamado de relacionamento recursivo ou autorrelacionamento.
- Binário (grau 2): mais comum.
- Ternário (grau 3): maior complexidade.
- Ou mais...

O grau de um relacionamento corresponde ao número de entidades que participam do relacionamento. Assim, temos relacionamentos unários, binários, ternários e assim sucessivamente, muito embora não seja comum desenvolver diagramas com relacionamentos com mais de três entidades envolvidas. Existem outros motivos para isso, que serão apresentados adiante. Por ora, vamos detalhar um pouco mais os graus de relacionamentos mais comuns.

Um relacionamento unário é utilizado para representar ocorrências de autorrelacionamento, isto é, um relacionamento entre ocorrências de uma mesma entidade. Nesse caso, o conceito de papel da entidade no relacionamento auxilia no entendimento e leitura do DER. Uma dica para ajudar a identificar esse tipo de relacionamento é buscar por atributos que indicam uma ocorrência da própria entidade. Esses atributos são chamados de ATRIBUTOS RELACIONANTES, e um exemplo pode ser visto na figura 2.14. Outra sugestão é dar o nome de papéis a esses relacionamentos recursivos.

Empregado: nome, sexo, **departamentoLotação**, supervisor...

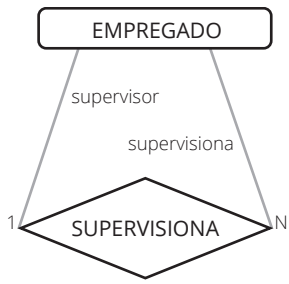


Figura 2.13
Exemplo de relacionamento recursivo ou autorrelacionamento.

Um relacionamento binário é aquele cujas ocorrências contêm duas ocorrências de entidade, como todos os vistos até aqui. Assim, podemos classificar os relacionamentos binários de acordo com a sua cardinalidade: N:N, 1:N e 1:1.

A abordagem ER permite que sejam definidos relacionamentos de grau maior do que dois (ternário, quaternário, e assim sucessivamente). No caso do relacionamento ternário, a cardinalidade refere-se a pares de entidades. Em um relacionamento R entre três entidades CIDADE, PRODUTO e DISTRIBUIDOR, a cardinalidade máxima de CIDADE e PRODUTO dentro de R indica quantas ocorrências de DISTRIBUIDOR podem estar associadas a um de ocorrências de CIDADE e PRODUTO. Exemplo na figura 2.5.

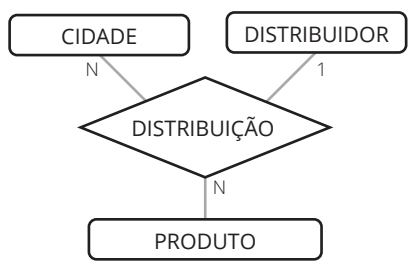


Figura 2.14
Exemplo de relacionamento ternário.

Relacionamentos com atributos

Uma pergunta comum para quem começa a trabalhar com DER é se relacionamentos podem ter atributos. A resposta é sim, pois podem existir dados que são inerentes ao fato que unem as respectivas entidades e não a qualquer uma das entidades. Alguns exemplos disso são:

- Empregado gerencia Departamento (data de início);
- Empregado trabalha em Projeto (número de horas);
- Paciente consulta Médico (data/hora);
  - Atributos podem ser omitidos no diagrama desde que especificados no Dicionário de Dados (DD) – ferramenta textual de apoio aos Diagramas;
  - Escolher nomes "apropriados".

Uma boa prática para melhorar a legibilidade dos DER é a escolha de nomes "apropriados", ou seja, que carreguem "significado" para ajudar no entendimento geral daquilo que está sendo representado no diagrama.

- Entidade: singular, maiúsculo (substantivo);
- Relacionamento: minúscula (verbo).

Ferramentas CASE

Ferramentas CASE (Computer-Aided Software Engineering) são ferramentas computadorizadas que contribuem/auxiliam no processo de engenharia de um software. Em outras palavras, são softwares cuja finalidade é ajudar no desenvolvimento de outros softwares.

Em nosso caso, estamos interessados em ferramentas CASE específicas para o processo de modelagem de sistemas de informação. É grande a quantidade de ferramentas desse tipo disponíveis no mercado, mas é preciso escolher aquela que será mais apropriada ao processo de modelagem sendo utilizado.

Observa-se que muitas dessas ferramentas não possibilitam a construção de modelos conceituais, partindo diretamente para o modelo lógico (e até físico) do banco de dados. Consideramos que isso não é desejável, pois descarta um passo importantíssimo, que é o foco desta sessão: a conceitualização do problema.

Descartar o Modelo Conceitual pode até ser admitido em sistemas pequenos, construídos por profissional com grande experiência em modelagem de dados, mas é inadmissível na fase de aprendizagem!

Exemplos de Ferramentas de Modelagem:

- DBDesigner;
- MySQL Workbench Design;
- PowerArchitect;
- Oracle's Designer;
- ERWin;
- brModelo.

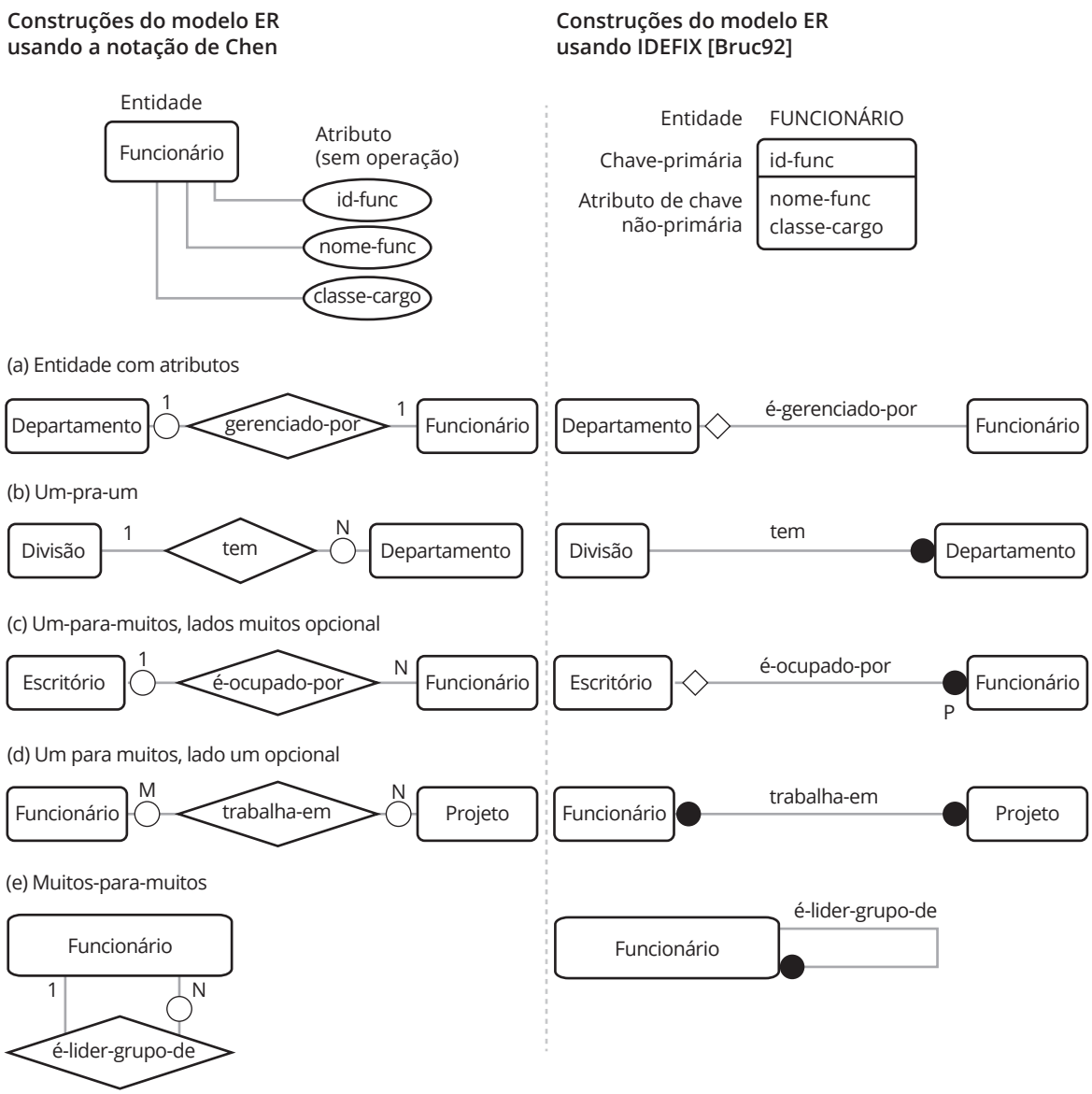
Notações

Já foi mencionado que diversas notações diferentes foram propostas em cima da notação originalmente idealizada por Peter Chein na década de 70. Uma representação de um modelo é chamada esquema do banco de dados. A figura 2.16 apresenta os principais símbolos utilizados na representação gráfica de um esquema ER, tanto na notação original como na notação IDEFIX proposta por Bachman.

A notação IDEFIX é a mais comumente implementada em ferramentas de modelagem, mas a verdade é que cada ferramenta pode trazer características específicas e pequenas (ou significativas) variações em relação a notação original.

Figura 2.15
Notações DER.

Neste curso, será utilizada a ferramenta brModelo, que será apresentada com mais detalhes a seguir.



A ferramenta brModelo

A brModelo é uma ferramenta freeware voltada para o ensino de modelagem em banco de dados relacional aplicando a notação original proposta por Chen com algumas adaptações. Assim, pode-se considerar a brModelo como uma ferramenta mais completa, já que a maioria das ferramentas disponíveis no mercado utilizam notações voltadas basicamente para a implementação do esquema gerado no SGBD, sem muita preocupação com o modelo lógico. Algumas dessas ferramentas sequer implementam o modelo conceitual.

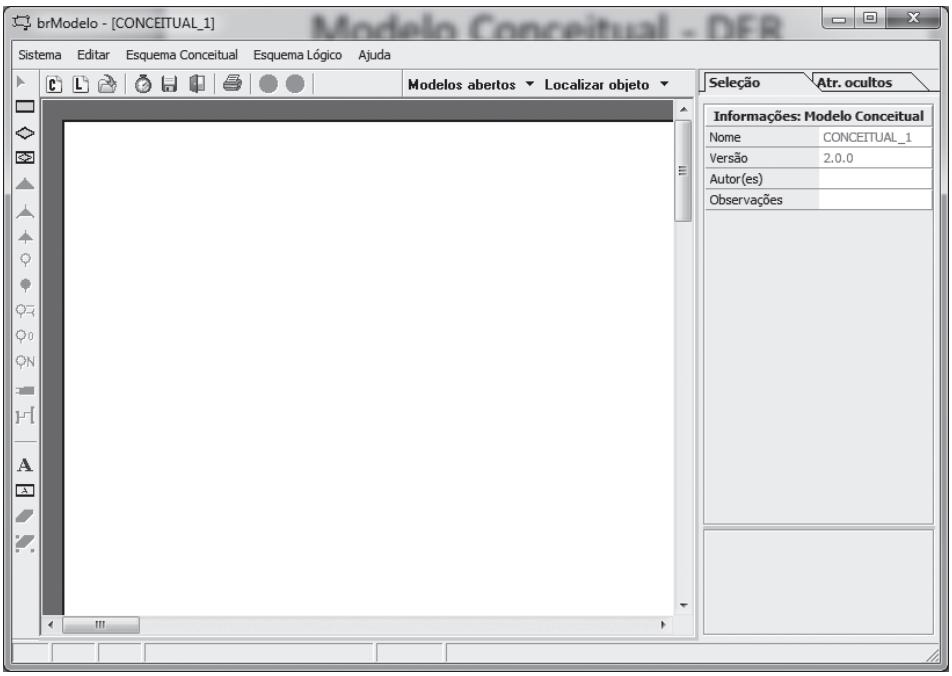


Figura 2.16
Interface gráfica do brModelo.

Saiba mais

Essa ferramenta foi desenvolvida por Carlos Henrique Cândido sob orientação do Prof. Dr. Ronaldo dos Santos Mello (UFSC), como trabalho de conclusão do curso de pós-graduação em banco de dados na UNVAG: MT e UFSC.

A figura 2.18 apresenta o DER do Mini-Mundo Empresa que foi apresentado anteriormente. Pode-se ver a qualidade dos gráficos gerados e como os conceitos até aqui apresentados são nela representados.

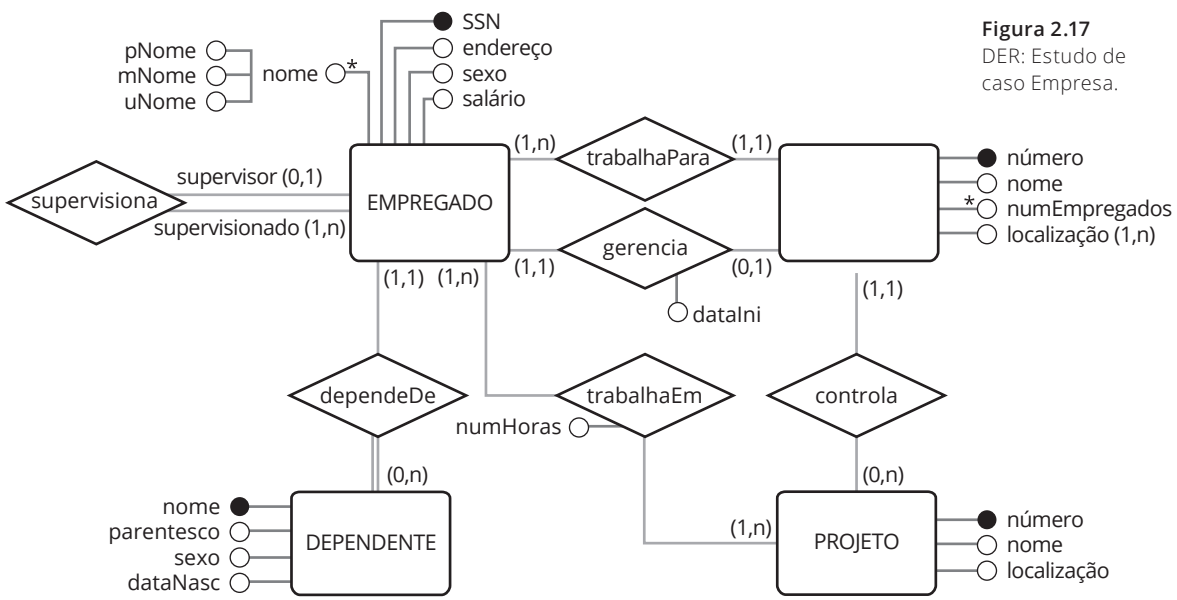


Figura 2.17
DER: Estudo de caso Empresa.

Atividade de Fixação 2.1

Refinamento do modelo conceitual

Técnicas de refinamento do modelo conceitual foram sendo implementadas para auxiliar no processo de modelagem de novas soluções de software que foram sendo desenvolvidas ao longo do tempo como, por exemplo, aplicações que passaram a utilizar o conceito de Orientação a Objetos, presentes em linguagens de programação a partir do final da década de 90.

No processo de refinamento do Modelo Conceitual, uma técnica bastante interessante é a de utilizar os conceitos de orientação a objetos, fazendo uso do Diagrama Entidade-Relacionamento Estendido: EER. Devemos lembrar que o modelo ER não suporta alguns conceitos semânticos necessários para modelar bases de dados mais recentes (GIS, CAD/CAM etc.).

O modelo EER é uma das alternativas ao modelo ER para modelar esse tipo de bases de dados.

Para isso, são trabalhados os seguintes conceitos:

- Herança;
- Multiplicidade: uso da cardinalidade: [mín, máx], CrowsFoot;
- Generalização-especialização (genespec)

Generalização/especialização

Associado ao conceito de generalização/especialização, está a ideia de herança de propriedades. Herdar propriedades significa que cada ocorrência da entidade especializada possui, além de suas próprias propriedades, as propriedades da ocorrência da entidade genérica correspondente. Assim, segundo o DER da figura 2.19 a seguir, a entidade PESSOA FÍSICA possui, além de seus atributos CPF e sexo, os atributos herdados da entidade CLIENTE (que são os atributos código e nome), bem como o relacionamento com a entidade FILIAL.

A especialização possibilita a atribuição de propriedades particulares a um subconjunto de ocorrências, ditas especializadas, de uma entidade genérica.

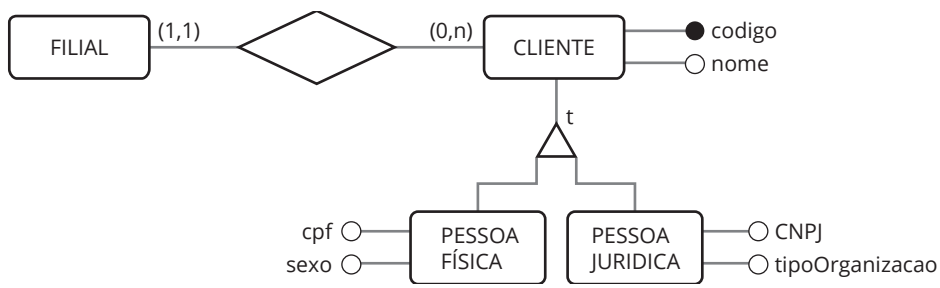
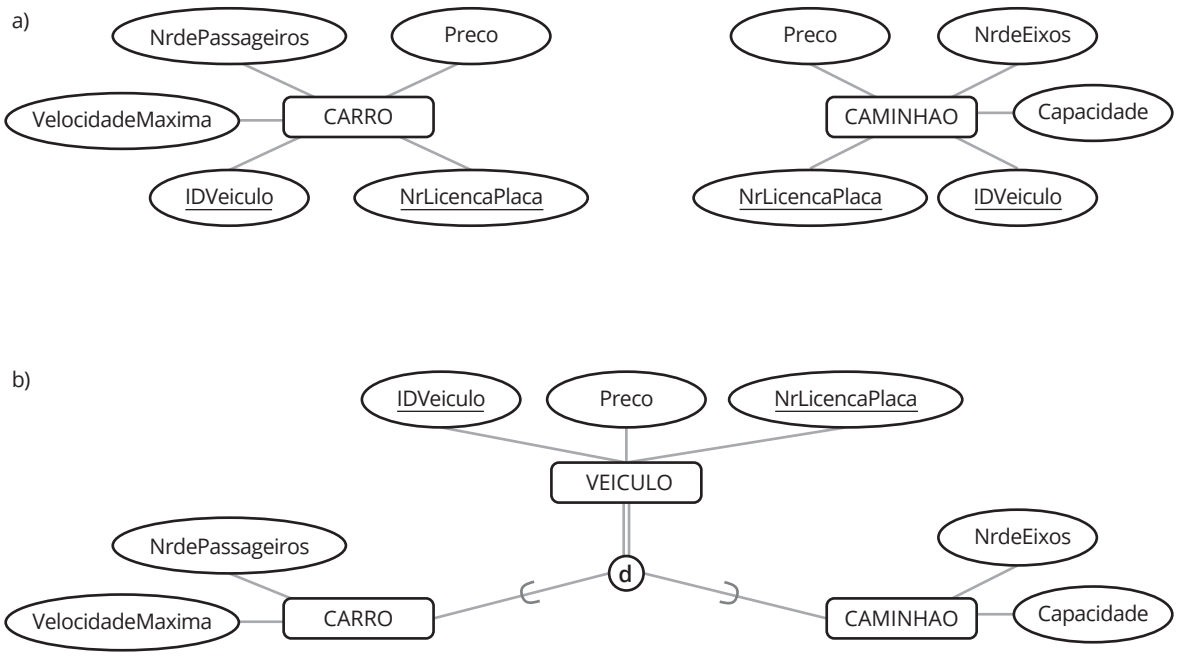


Figura 2.18 Exemplo de especialização.

Δ com base para as especializadas.

Como pode ser visto na figura 2.19, é utilizado um triângulo isósceles para representar o símbolo de especialização. Na figura 2.20, veremos que a generalização é representada através de um círculo seguido de linhas com o sinal de contido, reforçando quais entidades são a base para a identificação de uma generalização. Assim, temos a seguinte simbologia:



O com ⊂ para a genérica;

Figura 2.19
Exemplo de generalização.

É importante destacar que a árvore resultante do processo de herança, identificada pela generalização/especialização a ser modelada, deve ter uma única entidade raiz, sendo que esta deverá conter o identificador desse tipo de relacionamento. Na figura 2.21, apresentamos um exemplo ilustrando isso.

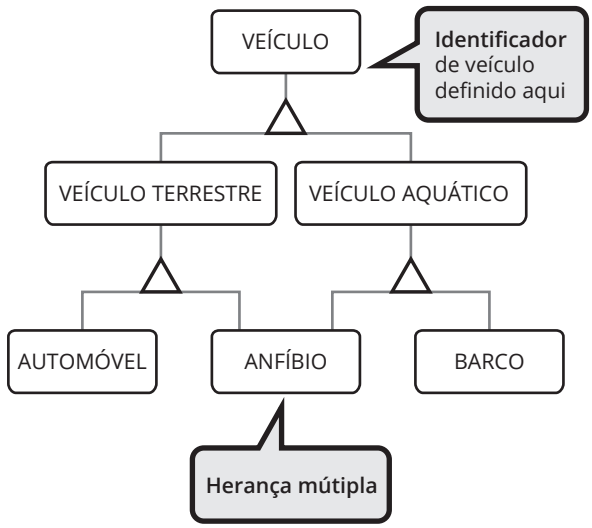


Figura 2.20
Exemplo de identificador único no processo de modelagem de herança.

Além disso, o identificador deve ser uma chave exclusiva que indica a relação existente entre a entidade especializada e a entidade genérica. Chamamos a atenção para a impossibilidade de modelagem de herança com múltiplos identificadores, conforme podemos observar na figura 2.22.

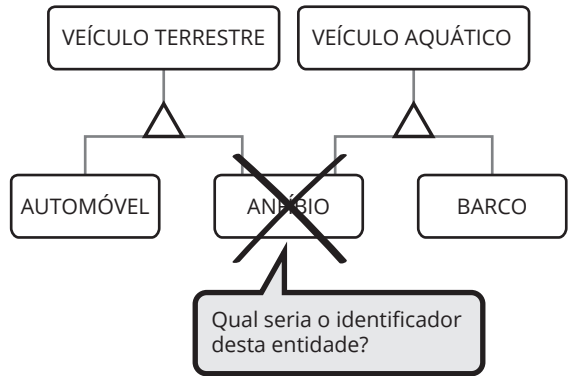


Figura 2.21
Hierarquia proibida: herança de vários identificadores.

Finalmente, na figura 2.23 apresentamos um exemplo demonstrando a possibilidade da recursão tanto na generalização como na especialização. Nesse exemplo, é apresentada a estrutura hierárquica de uma generalização.

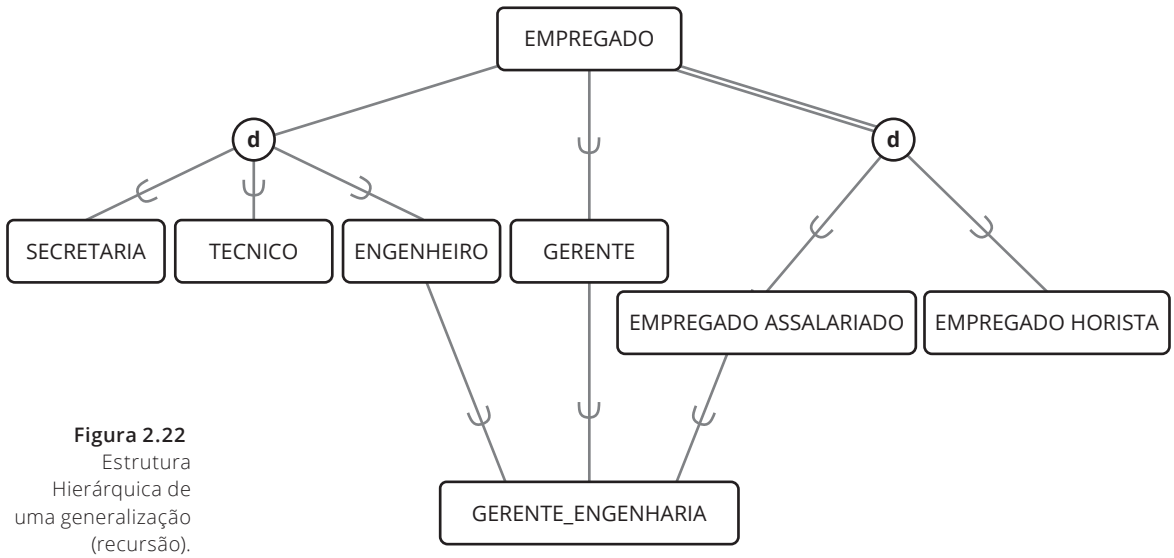


Figura 2.22
Estrutura Hierárquica de uma generalização (recursão).

Assim, de forma resumida, temos que:

- Especializadas herdam as propriedades da entidade genérica;
- Identificador somente na entidade genérica;
- Permite recursão (estrutura hierárquica): Não há limite no número de níveis;
- Permite Herança Múltipla (exemplo: ANFÍBIO);
- Uma entidade pode ser especializada em qualquer número de entidades, inclusive uma.

Tipos de generalização/especialização

A generalização/especialização pode ser classificada como:

- Total ou parcial: define a obrigatoriedade ou não de ocorrências da entidade genérica corresponder a uma ocorrência da entidade especializada.
- Exclusiva ou compartilhada: define se uma ocorrência da entidade genérica pode ser especializada uma ou mais vezes.

A seguir, analisamos cada um desses tipos separadamente.

Especialização Total (ou disjunta)

Na generalização/especialização total, cada ocorrência da entidade genérica deverá estar associada a uma ocorrência de uma de suas entidades especializadas. Esse tipo de especialização/generalização é simbolizado por um "t" junto ao triângulo isóscele que representa o relacionamento (generalização/especialização) entre as entidades.

Assim, para cada ocorrência da entidade genérica, existe sempre uma ocorrência em uma das entidades especializadas. Um exemplo de especialização total é apresentado na figura 2.24.

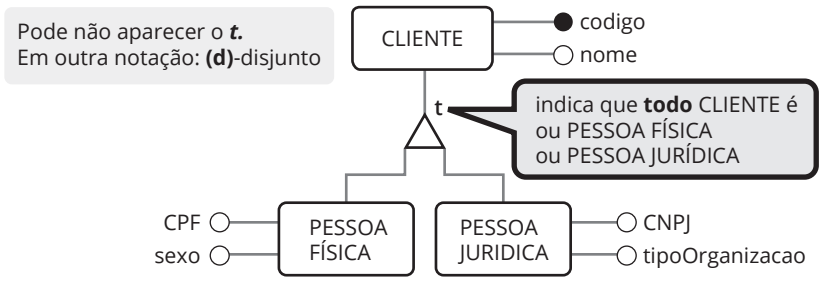


Figura 2.23
Exemplo de generalização/especialização total.

Especialização Parcial (ou notação linha simples)

Já na generalização/especialização parcial, nem toda ocorrência da entidade genérica possui uma ocorrência em uma de suas entidades especializadas. Esse tipo de especialização/generalização é simbolizado por um "p" junto ao triângulo isóscele, que representa o relacionamento (generalização/especialização) entre as entidades. A figura 2.25 apresenta um exemplo desse tipo.

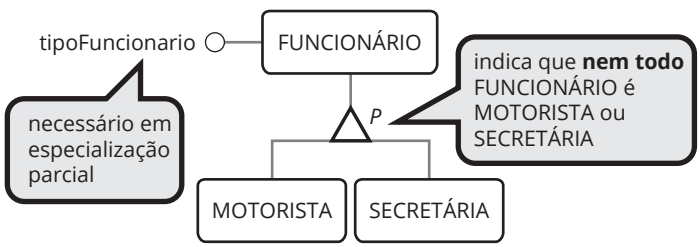


Figura 2.24
Exemplo de generalização/especialização parcial.

Especialização Exclusiva (x)

Na generalização/especialização Exclusiva, uma ocorrência de entidade genérica é especializada no máximo uma vez. Essa relação é representada pela notação utilizada na generalização/especialização total ou parcial, não sendo possível a ocorrência de mais de uma instância de entidade especializada para uma entidade genérica, conforme ilustrado nas figuras 2.24 e 2.25.

Compartilhada (c) ou Não Exclusiva

Na generalização/especialização Compartilhada, uma ocorrência da entidade genérica pode aparecer em várias de suas entidades especializadas. Esse tipo de especialização/generalização é simbolizado por um "c" junto ao triângulo isóscele que representa o relacionamento (generalização/especialização) entre as entidades. A figura 2.26 trás um exemplo desse tipo.

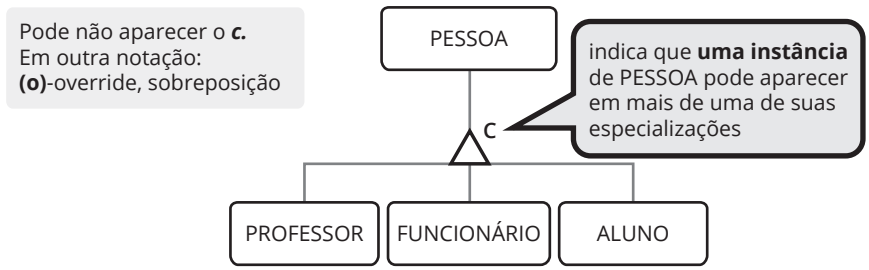


Figura 2.25 Exemplo de generalização/ especialização compartilhada.

De forma resumida, os diferentes tipos de generalização/especialização são apresentados na tabela da figura 2.27, onde as combinações de tipos de entidades são feitas utilizando as letras que as identificam no DER.

	Total (t)	Parcial (p)
Exclusiva (x)	xt	xp
Compartilhada (c)	ct	cp

Figura 2.26 Tipos de generalização/ especialização.

Entidade associativa

A entidade associativa nada mais é do que a redefinição de um relacionamento que passa a ser tratado como se fosse uma entidade.

Vale lembrar que na definição do modelo ER não foi prevista a possibilidade de se estabelecer relacionamentos entre relacionamentos, mas apenas entre entidades. Por outro lado, existem situações onde parece ser necessário associar uma entidade com a ocorrência de um relacionamento. Nesses casos, deve-se tratar esse relacionamento como se ele fosse uma entidade (entidade associativa). A notação utilizada para tanto é colocar um retângulo em torno do relacionamento (losango), conforme pode ser visto na figura 2.28.

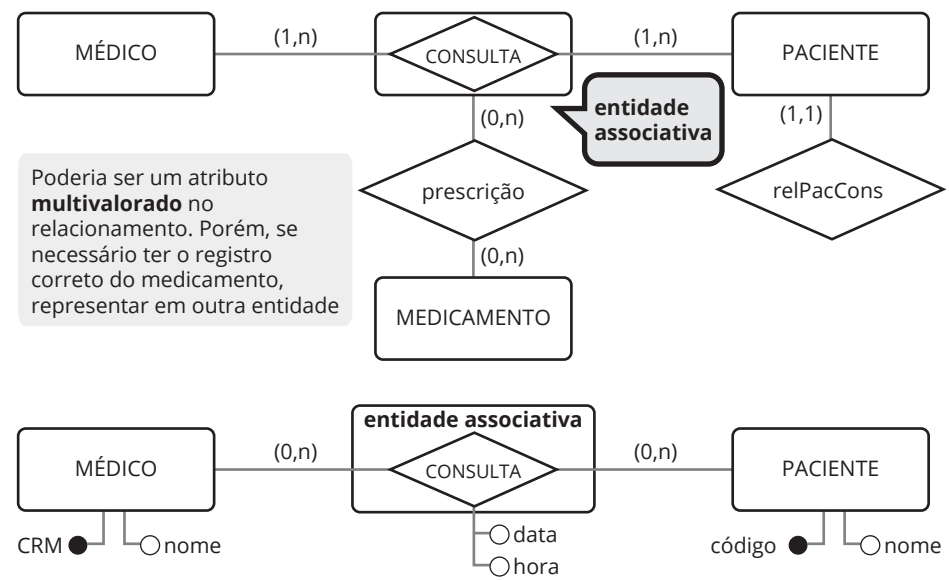


Figura 2.27 Exemplo de entidade associativa.

Outra solução possível, demonstrada na figura 2.29, é a substituição do relacionamento por uma entidade (caso não se deseje utilizar o conceito de entidade associativa).

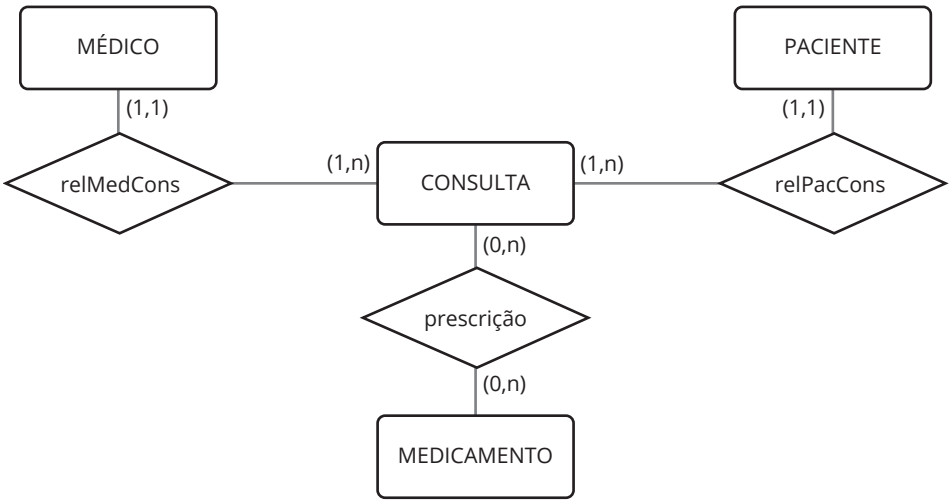


Figura 2.28
Exemplo de substituição de relacionamento por entidade.

Outro exemplo: adicionar a informação de que itens (quantidade, valor pago,...) de produto foram vendidos em uma venda. 'Entidade fraca' – N:N

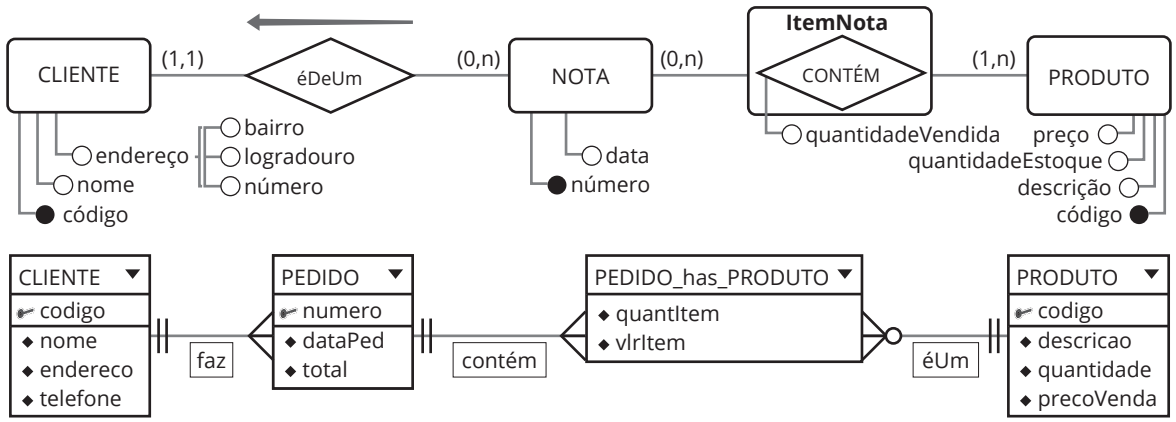


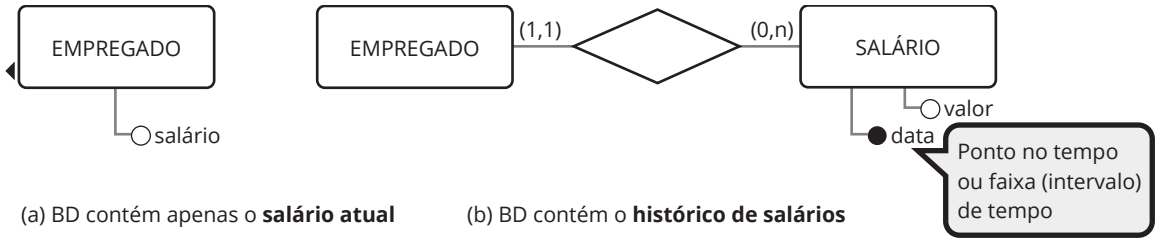
Figura 2.29
Outro exemplo de substituição de relacionamento por entidade.

Dados Temporais

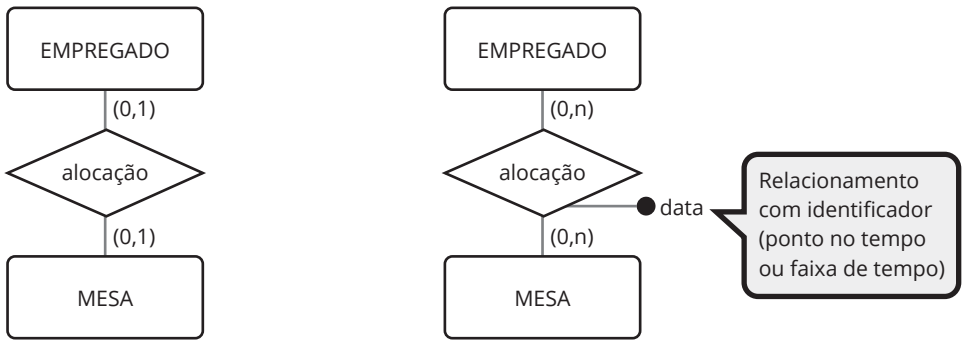
Um modelo deve se ocupar em refletir o aspecto temporal dos dados e informações que nele estão sendo modelados. Não existe uma regra geral de como proceder nesse caso, mas é possível identificar alguns padrões que se repetem, indicando uma possível necessidade de armazenamento temporal de dados e informações. De um modo geral, dados temporais têm as seguintes características:

- São dados que mudam ao longo do tempo.
  - Atributos cujos valores modificam ao longo do tempo;
  - Relacionamentos que modificam ao longo do tempo.
- São dados para os quais deve-se manter um histórico (motivos legais, auditorias, tomada decisão através de série temporal etc.).
  - Atributo de status (ativo/inativo);
  - 'desova' do BD (expurgo).

Na figura 2.30, apresentamos alguns exemplos de dados temporais.



(a) BD contém apenas o **salário atual** (b) BD contém o **histórico de salários**



(a) BD contém apenas a **alocação atual** (b) BD contém o **histórico de alocações**

Figura 2.30 Modelagem de dados temporais no DER.

Normalização

Em geral, a fonte de informações para análise e posterior construção do DER é o Minimundo produzido durante a análise de requisitos do sistema. Porém, os dados que serão analisados como ponto de partida para o DER podem estar armazenados em outras fontes de informação, como sistemas legados em bancos não relacionais, ou mesmo em bancos de dados relacionais, mas sem documentação.

Em uma situação como essa é interessante analisar os dados existentes – arquivos, tabelas e/ou formulários, e tentar construir o modelo conceitual (DER) correspondente. A motivação para isso é não somente gerar um modelo que possa auxiliar a compreender e documentar o sistema, mas principalmente identificar problemas de desempenho e integridade e, assim, facilitar a manutenção e evolução do banco de dados e dos sistemas que dele fazem uso.

A Normalização é a técnica que foi concebida para ser utilizada justamente em situações como essas. Ressaltamos que o processo de normalização pode ser executado sobre qualquer tipo de representação de dados: arquivo de computador, leiaute de relatório/formulário ou mesmo de uma tela.

Tomando como base um formulário qualquer, mas que contenha um grande número de campos, seria até possível pensar em colocar todos os dados do formulário em uma única grande tabela. Muito provavelmente, contudo, essa abordagem resultará em problemas de redundância de dados, que por sua vez implicam na geração de inconsistências nos valores dos dados e também no desperdício de espaço para armazenamento dos dados. Isso tudo acaba aumentando o tempo necessário para realizar a busca por informações.

Esses mesmos problemas podem já existir em dados armazenados em sistemas antigos, produzidos sem mais preocupações em relação à performance ou espaço de armazenamento. Mas o mais comum é que a demanda por novas funcionalidades acabe exigindo ajustes na estrutura do banco de dados que podem resultar em problemas como os recém-relatados. Assim, é preciso buscar desmembrar as informações e distribuí-las em tabelas menores, analisando as interdependências entre atributos individuais associados a cada uma dessas tabelas.

Ou seja, o processo de normalização pode ser entendido como um conjunto de regras que visa minimizar as anomalias em torno da modificação dos dados de modo que seja mais fácil manipular esses dados (manutenção do sistema) sem gerar redundâncias ou inconsistências. Resumidamente:

Por que normalizar?

- Minimização de redundâncias e inconsistências;
- Facilidade de manipulações do Banco de Dados;
- Facilidade de manutenção do Sistema de Informações.

Não custa lembrar que a normalização é aplicada em sistemas pré-existentes (sejam eles já computadorizados ou não). Para sistemas novos, um projeto bom e bem elaborado de um modelo conceitual resultará em bancos de dados que já estão normalizados ou que podem ser facilmente normalizados.

De qualquer modo, analisaremos a seguir as etapas de um processo de normalização.

Etapas normalização

A figura 2.32 trás um esquema descrevendo as diferentes etapas de um processo de normalização.

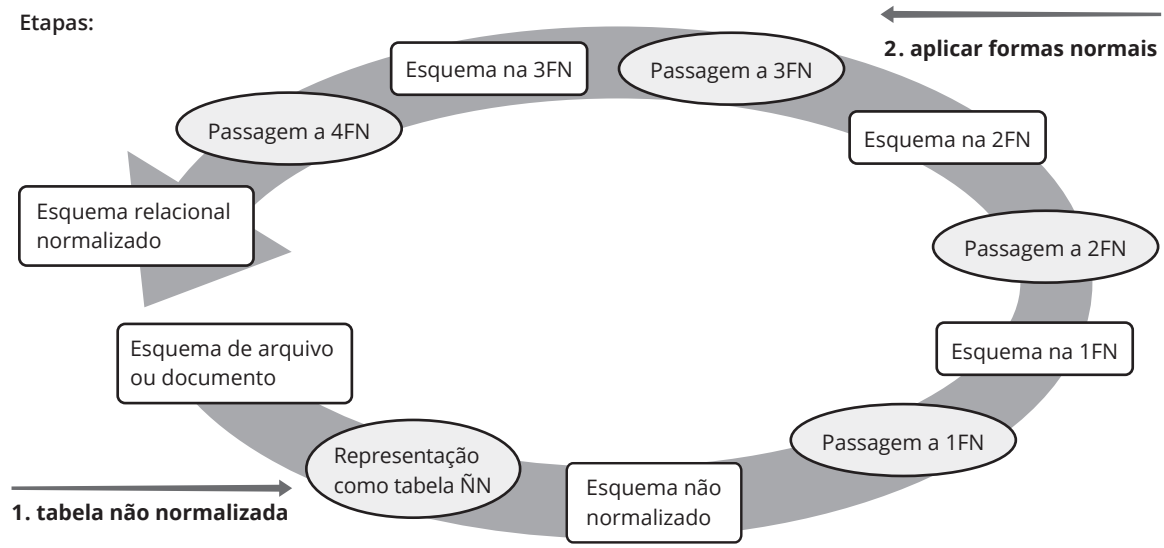


Figura 2.31
Etapas do processo de normalização.

É um processo cíclico, que deve ser usado sempre que um sistema precisa passar por um processo de manutenção.

O Passo 1 se inicia pela representação da descrição de cada arquivo ou documento existente na forma de um esquema de uma tabela relacional não normalizada (ÑN). O segundo passo é aplicar o processo de normalização a esse esquema de tabela não normalizada

Tabela não normalizada

Uma tabela não normalizada (ÑN) é um conjunto de dados no seu formato original, que é transposto para uma tabela relacional de uma forma "direta", sem ainda nenhuma preocupação com questões de redundância ou performance. Assim, na figura 2.33 temos um exemplo que ilustra uma tabela ÑN.

RELATÓRIO DE ALOCAÇÃO A PROJETO					
CÓDIGO DO PROJETO: LSC001			TIPO: Novo Desenv.		
DESCRIÇÃO: Sistema de Estoque					
CÓDIGO DO EMPREGADO	NOME	CATEGORIA FUNCIONAL	SALÁRIO	DATA DE INÍCIO NO PROJETO	TEMPO ALOCADO AO PROJETO
2146	João	A1	4	1/11/91	24
3145	Sílvio	A2	4	2/10/91	24
6126	José	B1	9	3/10/92	18
1214	Carlos	A2	4	4/10/92	18
8191	Mário	A1	4	1/11/92	12
CÓDIGO DO PROJETO: PAG02			TIPO: Manutenção		
DESCRIÇÃO: Sistema de RH					
CÓDIGO DO EMPREGADO	NOME	CATEGORIA FUNCIONAL	SALÁRIO	DATA DE INÍCIO NO PROJETO	TEMPO ALOCADO AO PROJETO
8191	Mário	A1	4	1/05/93	12
4112	João	A2	4	4/01/91	24
6126	José	B1	9	1/11/92	12

Figura 2.32
Exemplo de tabela não normalizada.

Ao fazer a transposição do conjunto de dados para uma tabela relacional, tomando como base o conjunto da figura 2.33, chegamos ao resultando apresentado na figura 2.34. Notem que temos a tabela relacional e também uma descrição textual dessa mesma tabela.

Códproj	Tipo	Descr	Emp					
			CodEmp	Nome	Cat	Sal	DataIni	TempAl
LSC001	Novo Desenvolvimento	Sistema de Estoque	2146	João	A1	4	01/11/91	24
			3145	Sílvio	A2	4	02/10/91	24
			6126	José	B1	9	03/10/92	18
			1214	Carlos	A2	4	04/10/92	18
			8191	Mário	A1	4	01/11/92	12
PAG02	Manutenção	Sistema de RH	8191	Mário	A1	4	01/05/93	12
			4112	joão	A2	4	04/01/91	24
			6126	José	B1	9	01/11/92	12

Figura 2.33
Tabela PROJETO representada de duas maneiras.

```
PROJETO (@cosProj, descrição, tipo,
                @codEmp, nome, cat, sal, dtIni, tAloc)
        )
```

Vejam que a tabela Projeto, não normalizada (ÑN), possui uma tabela aninhada dentro de si.

> Uma tabela aninhada é uma tabela que aparece como valor de campo dentro de outra tabela, condição que precisa ser ajustada.

Primeira forma normal (1FN)

A passagem para a 1FN pode ser feita pelo processo de decomposição de tabelas, levando-se em consideração os seguintes passos:

1. Criar uma tabela na 1FN que se refere a tabela não normalizada e que contém apenas as colunas com valores atômicos, sem as colunas de tabelas aninhadas
2. Criar uma tabela na 1FN para cada tabela aninhada, identificada na forma ÑN, com as seguintes colunas:
   2.1. A chave primária de cada uma das tabelas nas quais a tabela em questão está aninhada
   2.2. As colunas da própria tabela aninhada.
3. Identificar as chaves primárias das tabelas na 1FN que correspondem a tabelas aninhadas.

A figura 2.35 trás uma descrição da primeira forma normal (1FN), bem como a ilustração da transposição ÑN > 1FN.

primeira forma normal (1FN)
=
Diz-se que uma tabela está na primeira forma normal, quando ela não contém tabelas aninhadas

Passagem à primeira forma normal (1FN).

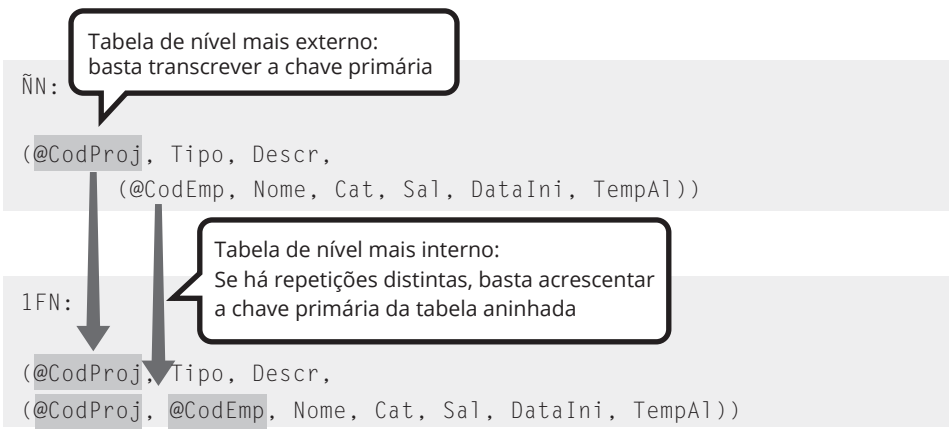


Figura 2.34
Forma ÑN para 1FN.

Dependência funcional

Antes de passarmos para as etapas seguintes do processo de normalização, é fundamental compreender o conceito de dependência funcional.

Um atributo ou conjunto de atributos A é dependente funcional de outro atributo B da mesma tabela, se a cada valor de B existir, nas linhas da tabela em que aparece, um único valor de A. Assim, A depende funcionalmente de B. A figura 2.36 trás um exemplo de uma dependência funcional.

...	B Código	...	A Salário	...
...	E1	...	10	...
...	E3	...	10	...
...	E1	...	10	...
...	E2	...	5	...
...	E3	...	10	...
...	E2	...	5	...
...	E1	...	10	...

Figura 2.35
Processo de identificação de dependência funcional entre atributos.

Salário depende funcionalmente do código do empregado.
Código → Salário

Na ocorrência de chave primária concatenada (mais de um atributo), a Dependência Funcional pode ser:

- Total: a cada valor completo da chave, está associado um valor de atributo;
- Parcial: quando o valor do atributo depende de parte da chave.
  - Observar cada entidade ou tabela com chave primária concatenada e analisar se existe algum atributo (ou conjunto de atributos) com dependência parcial.

Segunda forma normal (2FN)

Uma entidade encontra-se na 2FN, quando, além de encontrar-se na 1FN, cada uma de suas colunas que não são chave dependem da chave primária completa. Veja que toda tabela na 1FN que possui apenas uma coluna como chave primária já está na 2FN.

Os passos para se chegar a 2FN são:

1. Copiar para a 2FN cada tabela que tenha chave primária simples.
2. Para cada tabela com chave primária composta, considerando as colunas que não são parte da chave, fazer a pergunta: "A coluna depende de toda a chave ou de apenas parte dela?"
   - Se sim, as colunas permanecem na tabela original;
   - Se não, criar outra tabela na 2FN com parte da chave e as colunas da qual dependem.

Na figura 2.37, apresentamos um exemplo de normalização da 1FN para a 2FN.

segunda forma normal (2FN)
=
Uma tabela encontra-se na segunda forma normal,
quando, além de estar na 1FN, não contém *dependências parciais*

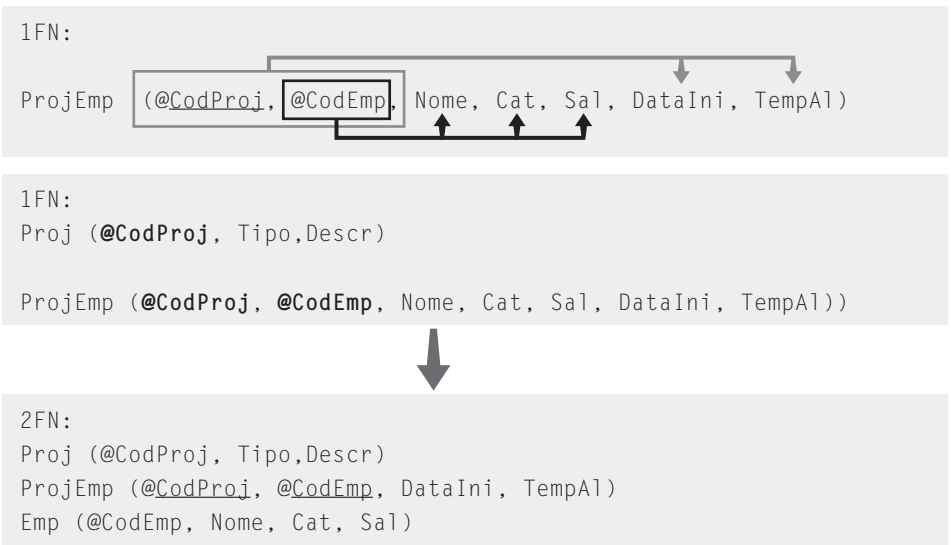


Figura 2.36
Da 1FN para 2FN.

Terceira forma normal (3FN)

Uma tabela encontra-se na 3FN quando, além de estar na 2FN, toda coluna não chave depende diretamente da chave primária, isto é, quando não há dependências funcionais transitivas ou indiretas.

Uma dependência funcional transitiva ou indireta acontece quando uma coluna não chave primária depende funcionalmente de outra coluna ou combinação de colunas não chave primária.

Assim, para chegar a 3FN, é preciso eliminar as dependências transitivas. Isso é feito da seguinte forma:

1. Colunas que dependem da chave primária permanecem na tabela original.
2. Colunas que dependem de coluna não chave vão para outra tabela.

Na figura 2.38, apresentamos a transformação da 2FN para 3FN.

terceira forma normal (3FN)
=
Uma tabela encontra-se na terceira forma normal, quando, além de estar na 2FN, não contém *dependências transitivas*

```
2FN:
Proj (@CodProj, Tipo,Descr)
ProjEmp (@CodProj, @CodEmp, DataIni, TempAl)
Emp (@CodEmp, Nome, Cat, Sal)
```

```
3FN:
Proj (@CodProj, Tipo,Descr)
ProjEmp (@CodProj, @CodEmp, DataIni, TempAl)
Emp (@CodEmp, Nome, Cat)
Cat (@Cat, sal)
```

Figura 2.37
Da 2FN para 3FN.

Quarta forma normal (4FN)

Para a maioria dos documentos e arquivos, a decomposição até a 3FN é suficiente para obter o esquema de um BD. Por outro lado, na literatura ainda encontramos a forma normal de Boyce/Code, que na verdade equivale a 4NF e 5FN.

Uma tabela estará na 4FN quando, além de estar na 3FN, não possuir mais que uma dependência funcional multivalorada. Em outras palavras, uma coluna ou conjunto de colunas dependem multivaloradamente de uma coluna determinante da mesma tabela quando um valor do atributo determinante identifica repetidas vezes um conjunto de valores na coluna dependente.

Para se chegar a 4FN, é necessário eliminar as dependências multivaloradas, o que equivale dizer que devemos eliminar eventuais relacionamentos ternários existentes no modelo. A figura 2.39 traz um exemplo disso.

quarta forma normal (3FN)
=
Uma tabela encontra-se na quarta forma normal, quando, além de estar na 3FN, não contém *dependências multivaloradas*

Dependência multivalorada
=
Uma dependência multivalorada ocorre quando um valor do atributo determinante identifica repetidas vezes um conjunto de valores na coluna dependente

```
3FN:
Util (@Codproj, @CodEmp, @CodEquip)
```

```
4FN:

ProjEmp (@CodProj, @CodEmp)
ProjEquip (@CodProj, @CodEquip)
```

Figura 2.38
Da 3FN para 4FN.

O processo de refinamento pode continuar com a 5FN, mas na prática os ganhos alcançados desse ponto em diante não justificam o trabalho envolvido. Para quem tiver interesse em conhecer a 5FN, sugerimos consultar o livro *Projeto e modelagem de banco de dados*, de Toby Teorey, Sam Lightstone e Tom Nadeau, da editora Elsevier.

Exercícios de Fixação
Refinamento do Modelo Conceitual - DER

Descreva com suas palavras. Por que devemos nos preocupar com a Normalização no processo de desenvolvimento de sistemas?

3

O Modelo Lógico

objetivos

Conhecer um conjunto de regras que auxiliam no processo de conversão do modelo conceitual em Modelo Lógico.

conceitos

Modelo Lógico; relações e restrições (de chaves e domínio); atributo multivalorado; tupla; chave primária; esquema.

O Modelo Lógico representa o banco de dados como uma coleção de relações. Informalmente, cada relação se parece com uma tabela. Dessa forma, podemos utilizar esse modelo para representar a estrutura de um Banco de Dados Relacional dentro de um SGDB(R).

Deve-se, portanto, construir o modelo lógico tomando como base o modelo conceitual que idealmente já terá sido validado pelos futuros usuários do sistema e demais atores envolvidos.

Por outro lado, ainda que não seja uma prática recomendada, sabemos que alguns desenvolvedores preferem pular a etapa de elaboração do modelo conceitual. Nesse caso, não haverá um modelo conceitual para servir de base para a elaboração do modelo lógico. Mais do que isso, algumas decisões de modelagem não terão ainda surgido, e o profissional pouco experiente poderá ser confrontado com situações que costumam gerar dúvidas.

Por conta disso, antes de passarmos a tratar diretamente do modelo lógico, vamos revisitar questões comuns que se colocam a todos os envolvidos com o processo de modelagem de um sistema de informações.

Decisões sobre o modelo

Somente entidade tem atributos?

Conforme já mencionado anteriormente, nada impede que um relacionamento tenha atributos. Por outro lado, em vez de desenvolver o modelo desse modo, uma alternativa que deve ser considerada é a criação de uma entidade associativa. A figura 3.1 a seguir traz um exemplo como este:

- Empregado trabalhaEm Projeto (numHoras);
- NotaFiscal contémItensDe Produto (quant, preçoUnit).

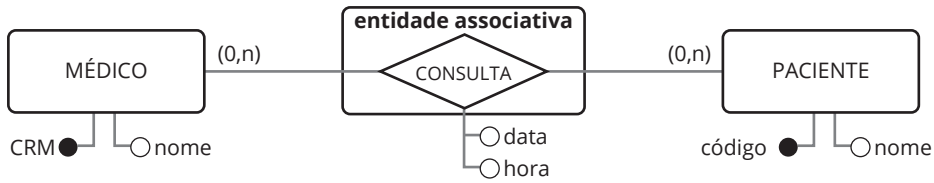


Figura 3.1
Decisão de se utilizar entidade associativa.

Atributo multivalorado?

Também é possível desenvolver um modelo onde uma ou mais entidades tenham atributos multivalorados. Mas o mais apropriado, especialmente quando a quantidade de ocorrências desse atributo não for muito grande, é preferir a criação de uma entidade relacionada para armazenar tais atributos. Na figura 3.2, temos mais de um exemplo desse tipo.

- Cliente com vários Telefones;
- Departamento em várias Localizações.

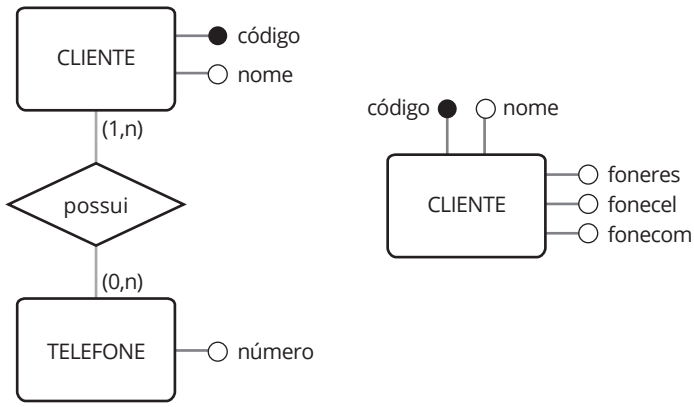
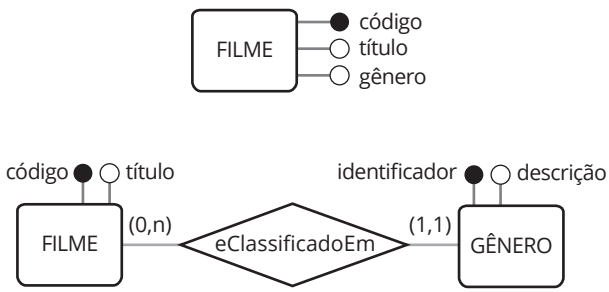


Figura 3.2
Decisão entre entidade ou atributo multivalorado.

Quando transformar um atributo em entidade?

Determinados atributos de uma entidade podem ser modelados como uma entidade autônoma relacionada a essa entidade. Conforme o profissional vai ganhando experiência, essa decisão torna-se cada vez mais fácil e "automática". Mas para quem ainda está começando, a tabela na figura 3.3 pode ser uma boa referência.



Situações a considerar	Uso
Se este atributo está vinculado a outros objetos no diagrama	Entidade
Desempenho de buscas	Atributo
Integridade da informação	Entidade
Custo de armazenamento	Entidade

Figura 3.3
Quando transformar um atributo em entidade?

Relacionamentos

As recomendações que devem ser observadas na modelagem de relacionamentos são as seguintes:

- N:N: transformar em entidade associativa;
- 1:1: em dependência obrigatória/total com cardinalidade mínima 1, optar por:
  - Substituir por uma única entidade;
  - Criar uma entidade associativa (histórico).

Outro ponto que deve ser observado, já pensando no modelo lógico, é utilizar somente relacionamentos binários entre as entidades do seu modelo. Assim, caso se depare com um relacionamento ternário, a recomendação é utilizar uma entidade associativa, conforme podemos observamos na figura 3.4:

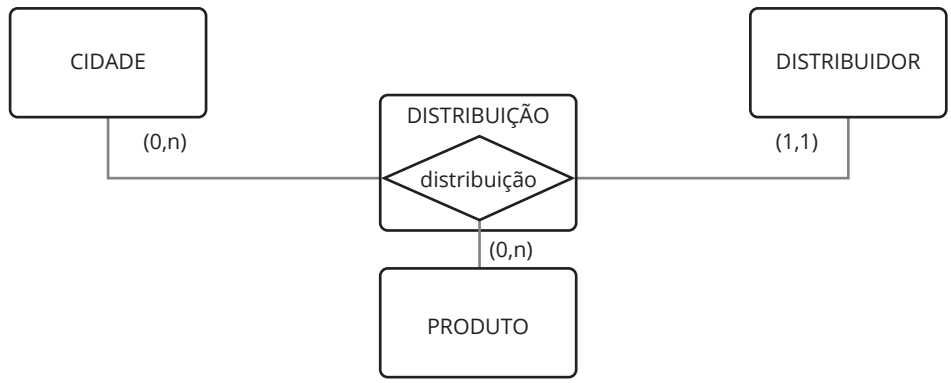


Figura 3.4 Conversão de relacionamento ternário em entidade associativa.

Já no caso de autorrelacionamentos (relacionamento unário) existem duas opções para manter apenas relacionamentos binários, conforme pode ser visto na figura 3.5.

- Criação de entidade representativa do "papel";
- Uso de generalização ou especialização (EER).

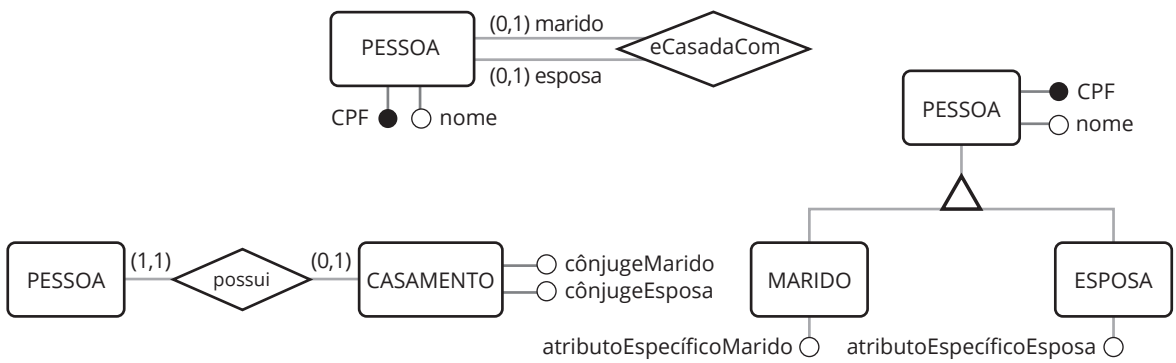


Figura 3.5 Transformando relacionamento unário em binário.

Uma última dica sobre a modelagem de relacionamentos se refere a entidades que possuem atributos particulares e que devem ser modelados através de genespec (EER). Exemplos dessa situação é quando temos de colocar no modelo tanto carros como motos, ambos veículos automotores mas com características específicas. Um outro exemplo muito comum é a modelagem de pessoa física versus pessoa jurídica. Isso pode ser melhor entendido através da figura 3.6.

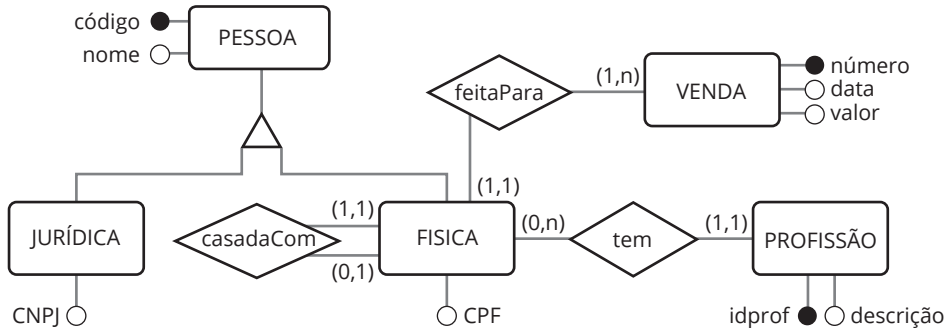


Figura 3.6
Exemplo de modelagem de atributos opcionais de entidades com EER.

Coleção de Relações

Conforme já foi mencionado no início desta sessão, o Modelo Lógico representa o banco de dados como uma coleção de relações (tabelas), se aproximando da estrutura que o banco de dados terá em um SGBD(R).

A base matemática se traduz pelo uso da teoria de conjuntos e da lógica de predicados de primeira ordem. Assim, uma tabela é um conjunto de tuplas, sendo cada tupla entendida como uma instância de uma "entidade". Outro ponto a ser observado é que todas as tuplas em uma relação são distintas (chave primária). A figura 3.7 reforça os conceitos acima descritos.

- Uma tabela é um conjunto de tuplas;
- Tupla: cada instância de uma "entidade";
- todas as tuplas em uma relação são distintas (chave primária).

CATEGORIA

codcat	nomecat
1	LIMPEZA
2	VESTUARIO
3	LATICINIO
4	ACOUGUE
5	FARINACEO
6	INFORMATICA

PRODUTO

codprod	descprod	precoprod	codcat
1	BOMBRIL	2,3	1
2	DESINFETANTE	2,95	1
3	SABÃO EM PÓ	5,15	1
4	CALÇA	35,9	2
5	CAMISETA	19,9	2
6	COXÃO MOLE	18,9	4
7	MOÍDA DE 1A	9,9	4
8	VESTIDO	45,8	2
9	FILÉ MIGNON	23,4	4
10	MONITOR LCD	380	6
11	HD EXTERNO	250	6
12	PEN DRIVE 8 GB	35	6
13	MOUSE USB	18,5	6
14	YAKULT	4,98	3
15	IOGURTE DANONE COM 6	2,68	3
16	CEREAL MATINAL	5,15	5

Figura 3.7
Ocorrência de tuplas em uma tabela.

Saiba mais

Esse modelo foi proposto em 1970 por Ted Codd, sendo rapidamente aceito em função de sua simplicidade e, ao mesmo tempo, base matemática (relações e teoria de conjuntos).

Outra forma de representar as tabelas que aparecem na figura 3.7, e que já foi utilizada na sessão anterior, é a notação descritiva que aparece na figura 3.8. Essa figura ajuda a entender os conceitos de Esquema de Relação e Esquema de Banco de Dados.

- Esquema de Relação:
  `R(A1, A2,..., An)`
- Esquema de um Banco de Dados:
  `S = {R1, R2,..., Rm} + conjunto RI`

Categoria (@**codCat**, descrCat)
Produto (@**codProd**, descrProd, precoProd, codCat)

Chave primária: @, **Chave estrangeira**: sublinhando, **Domínio**: no Dicionário de Dados

Figura 3.8
Esquema de Relação.

O Esquema de Relação é descrito pela notação R(A1, A2,..., An), onde R é o nome da relação ("entidade" derivada do DER) e Ai é cada um de seus atributos. Assim, podemos entender R como sendo igual a Categoria, composta, respectivamente, pelos atributos A_1 = @codCat e A_2 = descrCat.

O esquema de relação pode ser interpretado como uma declaração ou um tipo de asserção. Por exemplo, o esquema da relação Categoria declara que, em geral, uma entidade categoria tem um código e sua descrição. Cada tupla na relação pode ser interpretada como um fato ou uma instância em particular da asserção.

Já o Esquema de Banco de Dados é representado por um conjunto de esquemas de relação através da notação S = {R1, R2,..., Rm}. No exemplo da figura 3.8, o banco de dados em questão é composto pelas relações R_1 = Categoria e R_2 = Produto.

A figura 3.8 faz menção também a um conjunto RI. Vejam que o esquema de banco de dados é utilizado para verificação do estado de um banco de dados relacional de esquema S, se preocupando em verificar se os estados de uma determinada relação satisfazem às restrições de integridade especificadas no conjunto RI. Dessa forma, um estado de um banco de dados que não obedece a todas as restrições de integridade é chamado de estado inválido, e um estado que satisfaz todas as restrições em RI é conhecido com estado válido.

Restrições de Integridade

O Modelo lógico, em função de sua base matemática, deve observar as seguintes restrições de integridade:

- Restrição de Chave:
  - Chave primária (@Primary Key);
- Restrição de Integridade Referencial:
  - Chave estrangeira (Foreign Key);
- Restrição de Domínio:
  - Tipo do dado: representado no Dicionário de Dados.

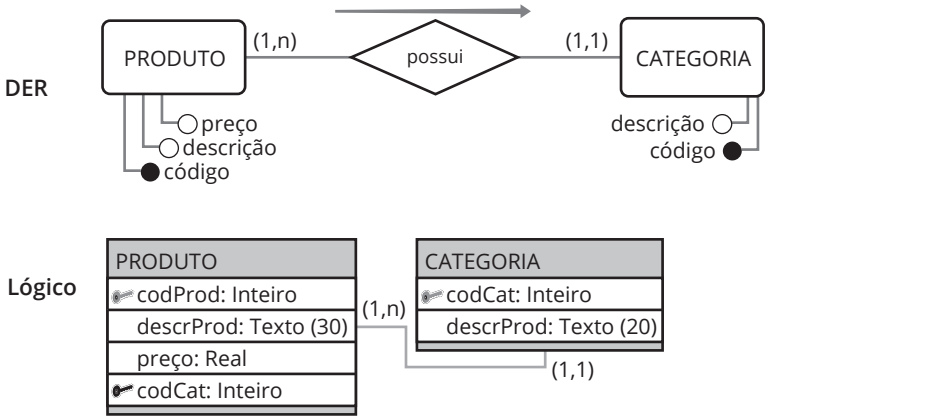
A Restrição de Chave preconiza que o valor de cada chave primária deve ser único para todas as tuplas de qualquer relação do esquema (e não podem receber valor nulo). Essa restrição garante a identificação unívoca de cada tupla em uma relação.

A Restrição de Integridade Referencial faz uso das chaves estrangeiras, já que demanda que uma tupla de uma relação que se refere a outra relação deve sempre se referir a uma tupla existente naquela outra relação. Essa restrição é classificada entre duas relações e é usada para manter a consistência entre as tuplas nas duas relações.

Finalmente, a Restrição de Domínio demandada especifica que, dentro de cada tupla, o valor de cada atributo A deve ser um valor atômico do domínio dom(A). Os tipos de dados associados aos domínios incluem os tipos de dados numéricos para representação de inteiros e números reais, caracteres, booleanos, datas, horas, timestamp e, em alguns casos, os tipos de dados de moeda.

Modelo Conceitual (DER) x Modelo Lógico

A construção do esquema de banco de dados deve ser orientada por um conjunto de regras de auxiliam no mapeamento do Modelo Conceitual (DER) para o Modelo Lógico, mais próximo da estrutura do banco de dados em um SGBD(R). Esse desafio é resumido pela figura 3.9.



Categoria (@**codCat**, descrCat)
Produto (@**codProd**, descrProd, precoProd, codCat)

Chave primária: @, **Chave estrangeira**: sublinhando, **Domínio**: no Dicionário de Dados

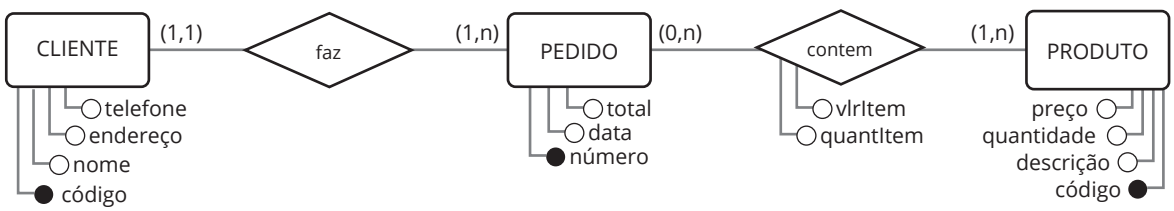
Figura 3.9
Conversão para modelo lógico.

Nesse processo, vamos usar as seguintes regras para conversão dos construtores básicos do modelo ER em relações:

- Entidades e atributos;
- Relacionamentos binários (1:1, 1:N, N:N);
- Relacionamento *n*-ário;
- Relacionamento Unário ou recursivo;
- Generalização ou especialização;
- Atributo multivalorado.

Cada uma dessas regras será explorada em maior detalhe a seguir. Após a aplicação de todas as regras propostas por esse mapeamento teremos, então, a especificação do banco de dados. A figura 3.10 trás, como exemplo, o resultado da aplicação das regras acima na conversão de um DER para um conjunto de relações.

Figura 3.10
Exemplo do uso das Regras de Mapeamento.



CLIENTE (@**codCli**, nomeCli, enderCli, foneCli)
PEDIDO (@**numPed**, codCli, dataPed, totalPed)
PRODUTO (@**codProd**, descProd, quantProd, precoVendaProd)
ITEM_PEDIDO (@**numPed**, @**codProd**, quantItem, vlrItem)

Entidades e atributos

Essa regra deve ser aplicada para cada entidade no DER.

A primeira providência para uma entidade forte E é criar um esquema de relação R que inclua todos os atributos de E. Em seguida, escolha um dos atributos identificadores de E como chave primária de R. A aplicação dessa regra é demonstrada na figura 3.11, inclusive a preocupação com a eventual decomposição de atributos compostos.

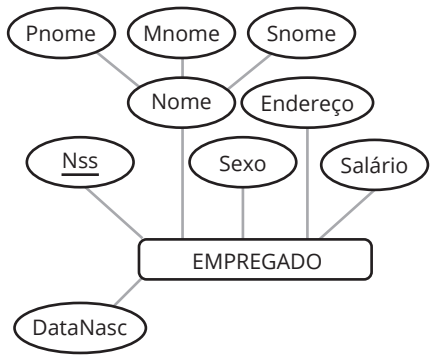


Figura 3.11
Regra Entidades (Fortes) e Atributos.

Exemplo:
EMPREGADO (@**SSN**, pNome, inicialM, uNome, sexoEmp, endereço, salário, dataNascEmp)

Já para cada entidade fraca F no DER, a mesma providência inicial deve ser tomada: crie um esquema de relação R que inclua todos os atributos F. Em seguida, inclua como chave estrangeira de R os atributos que formam a chave primária do esquema de relação associada à entidade forte E de F. A chave primária de R será a combinação da chave primária de E com a chave parcial de F. Isso é melhor ilustrado na figura 3.12.

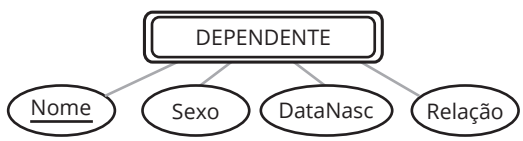


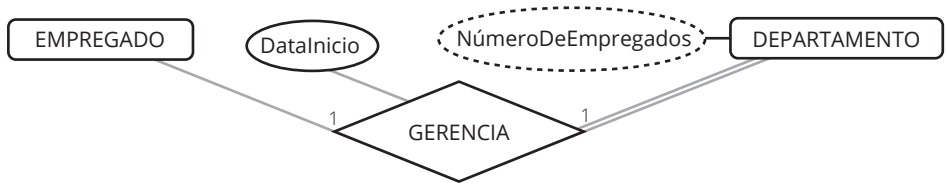
Figura 3.12
Regra Entidades (Fracas) e Atributos.

Exemplo:
EMPREGADO (@**SSN**, pNome, inicialM, uNome, sexoEmp, endereço, salário, dataNascEmp)
DEPENDENTE (@**SSN**, @**Nome**, SexoDep, dataNascDep, parentesco)

Relacionamentos binários

A regra para a conversão de Relacionamentos binários tem variações para cada uma das possibilidades: 1:1, 1:N e N:M.

Para os relacionamentos 1:1, é preciso inicialmente identificar os esquemas de relação S e T que dele participam. Esses devem ser combinados em uma só relação, mas não sem antes verificar se são concebidos separadamente no ambiente de negócio ou se há cardinalidade mínima zero. Isso evitará desperdício de espaço. Outra alternativa é escolher um dos relacionamentos, digamos S, e nele incluir como chave estrangeira a chave primária de T (se houver atributos simples no relacionamento, inclua-os como atributos de S).
A figura 3.13 ilustra a aplicação dessa regra.



Exemplo:
EMPREGADO (@**SSN**, pNome, inicialM, uNome, sexoEmp, endereço, salário, dataNascEmp)
DEPARTAMENTO (@**NúmDepto**, nomeDepto, numEmp, **SSNChefe, dataInício**)

Figura 3.13
Regra Relacionamentos binários (1:1).

Já para o caso de relacionamentos 1:N, é preciso identificar o esquema de relação S que representa o tipo de entidade do "lado n" do relacionamento. Feito isso, inclua como chave estrangeira de S a chave primária do esquema T que representa o outro tipo de entidade ("do lado 1"). Se houver atributos simples no relacionamento, inclua-os como atributos de S. A figura 3.14 exemplifica essa conversão.

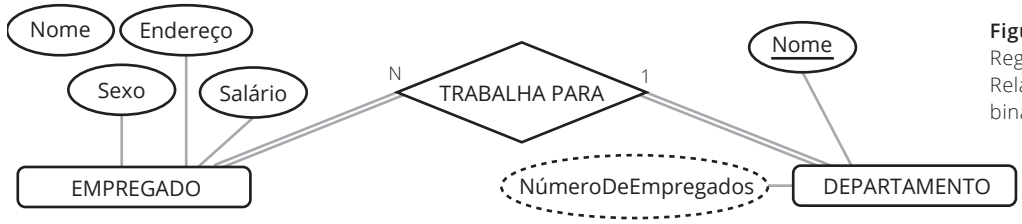
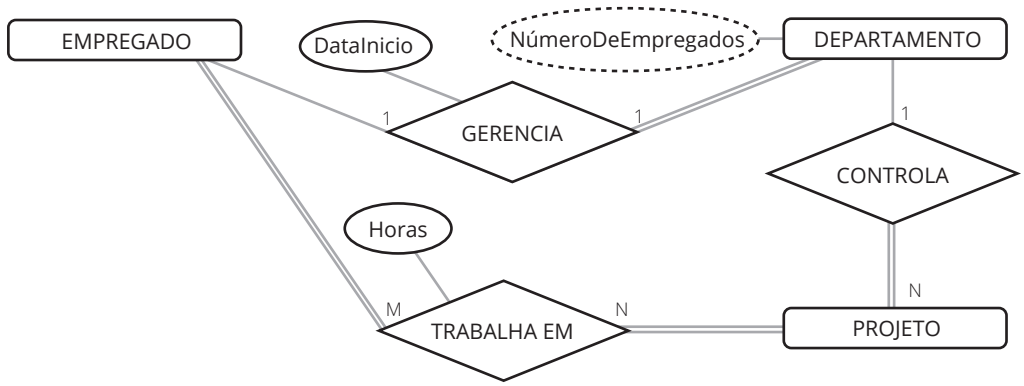


Figura 3.14
Regra Relacionamentos binários (1:N).

Exemplo:
EMPREGADO (@**SSN**, pNome, inicialM, uNome, sexoEmp, endereço, salário, dataNascEmp, **NúmeroDepto**)
DEPARTAMENTO (@**NúmDepto**, nomeDepto, numEmp, SSNChefe, dataInício)

Por fim, ao lidar com relacionamentos N:N, o passo inicial é criar um novo esquema de relação S para representá-lo. A chave estrangeira de S será composta pelas chaves primárias dos esquemas de relações que representam os tipos de entidade participantes no relacionamento N:N. Se houver atributos simples no relacionamento, inclua-os como atributos de S. Tenha em mente que podem não existir relacionamentos N:N no DER se foram utilizadas entidades associativas. Por isso é que se recomenda seu uso, deixando o DER mais próximo do futuro modelo conceitual. De qualquer modo, a figura 3.15 traz um exemplo da aplicação dessa regra.

Figura 3.15
Regra Relacionamentos binários (N:N).



Exemplo:
EMPREGADO (@**SSN**, pNome, inicialM, uNome, sexoEmp, endereço, salário, dataNascEmp, **NúmeroDepto**)
PROJETO (@**NumProj**, NomeProj, Localização)
TRABALHA_EM (@**SSN**, @**NumProj**, Horas)

Relacionamento *n*-ário, onde *n* > 2

Relacionamentos "maiores" do que relacionamentos binários devem ser convertidos através da criação de um novo esquema de relação S para representá-los. A chave estrangeira dessa nova relação S será composta pelas chaves primárias de todas as entidades que participam do relacionamento original. Se houver atributos simples nos relacionamentos, inclua-os como atributos de S. Mais uma vez, poderão não existir relacionamentos desse tipo no DER, especialmente se foram utilizadas entidades associativas ou se o modelo conceitual tiver passado por um processo de normalização. A figura 3.16 apresenta um exemplo da aplicação da regra acima.

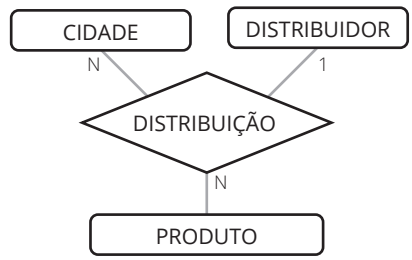


Figura 3.16
Regra Relacionamento n-ário, onde n > 2.

Exemplo:
DISTRIBUIÇÃO (@codDist, @codCid, @codProd, ...)

Relacionamento unário (ou recursivo)

Essa regra admite três alternativas de conversão:

- Criar um novo atributo para representar o "papel" no relacionamento (1:1).
  - Pessoa casada com Pessoa-cônjuge.
- Criar um novo esquema para representar o relacionamento(1:N).
  - Vendedor é gerente de Vendedor: relação Gerência (numGerente, numGerenciado)

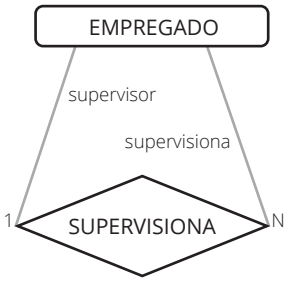
Partir para uma generalização ou especialização (quando existem atributos diferentes a serem considerados), seguindo a próxima regra.

A primeira alternativa é criar um novo atributo no esquema de relação S para representar o "papel" no relacionamento. É o mais recomendado quando o relacionamento recursivo é 1:1.

A segunda alternativa, mais recomendada para relacionamentos 1:N, é criar um novo esquema de relação S para representar o relacionamento. É preciso verificar a cardinalidade do relacionamento e seguir como definido para relacionamentos binários.

A terceira e última alternativa é partir para uma generalização ou especialização, em geral quando existem atributos diferentes a serem considerados. Nesse caso, deve-se seguir a próxima regra, que trata de generalizações e especializações.

De qualquer modo, na figura 3.17 apresentamos um exemplo de conversão de relacionamento unário.



Exemplo:
EMPREGADO (@**SSN**, pNome, inicialM, uNome, sexoEmp, endereço, salário, dataNascEmp, NúmeroDepto)
SUPERVISÃO (@**SSNsupervisor**, @**SSNsupervisonado**)

Figura 3.17
Conversão de relacionamento unário.

Generalização ou especialização

A primeira providência é verificar se existem relacionamentos com a entidade geral. Em caso afirmativo, deve-se primeiro criar um esquema de relação L para a entidade geral C (superclasse), com atributos {k, a_1, a_2,..., a_n}, sendo k o atributo identificador. Em seguida, crie uma relação L_i para cada entidade específica Si (subclasses), onde os respectivos atributos serão k ∪ {atributos de S_i } (k é a chave primária). Finalmente, para facilitar o processo de pesquisa no banco de dados, recomenda-se a criação de um atributo "tipo" no esquema de relação L para identificar a subclasse a que os dados se referem. Essa situação pode ser representada por:

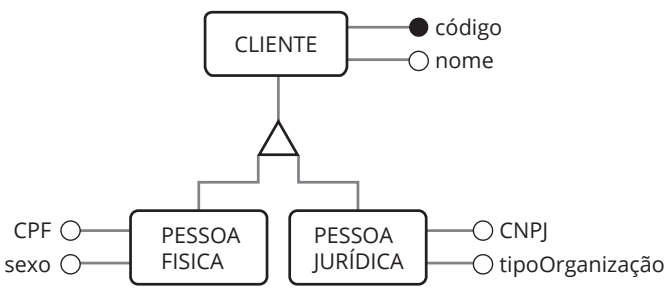
`L{k, a1, a2,..., an, t}` – sendo k o atributo identificador e t é um atributo "tipo".

`Li {k ∪ {atributos de Si}}`

Por outro lado, se não existe relação com a entidade geral, deve-se então criar uma relação L_i para cada entidade específica S_i (subclasses). Os atributos de cada relação L_i serão a união dos atributos com os atributos da entidade geral C (superclasse), incluindo obviamente o seu identificador/chave. Isso pode ser representado pela expressão:

`Li {{k, a1, a2,..., an} ∪ Si }` – sendo k a chave primária de C.

Na figura 3.18 a seguir demonstramos a aplicação dessa regra no caso em que há relacionamento com a entidade geral.



Exemplo:
CLIENTE (@**código**, nome, ***tipo***)
PFISICA (@**código**, CPF, sexo)
PJURÍDICA (@**código**, CNPJ, tipoOrganização)

Figura 3.18
Regra Generalização ou especialização.

Atributo multivalorado

Supondo o esquema de relação R{k, a_1, a_2, A,..., a_n}, onde A é o atributo multivalorado. Se o número de ocorrências de A é fixo, deve-se criar um novo atributo em R para cada uma dessas ocorrências de A.

Por outro lado, se a quantidade de ocorrências de A é muito grande, ou variável, deve-se então criar um novo esquema de relação NR, incluindo neste apenas dois atributos: um correspondente ao atributo multivalorado A e o outro correspondente à chave primária k de R. A chave primária de NR será a combinação de k e A. A seguir, exemplos para ambas as situações:

Considerando cliente e telefone, onde:

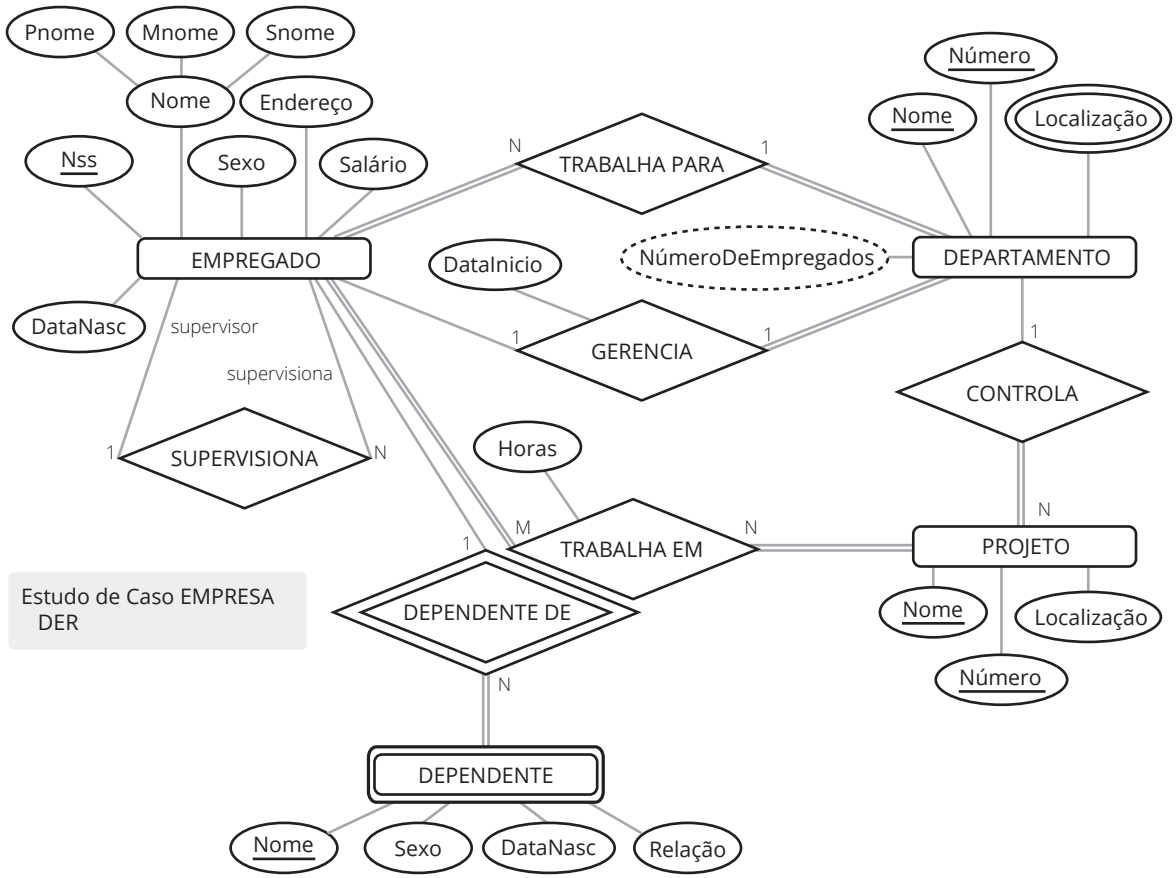
- O cliente tem sempre um telefone fixo, um celular e um telefone comercial:
  - Cliente(@codcli,..., FoneFixo, FoneCel, FoneComercial)
- O cliente pode ter uma quantidade ilimitada de telefones:
  - Telefone(@codcli, @numfone)

Quadro resumo

Modelo ER	Modelo Relacional
Entidade	Relação
Relacionamento 1 : 1 ou 1 : *N*	Relação + **uma chave** estrangeira
Relacionamento *M* : *N*	Relação + **duas chaves** estrangeiras
Atributo composto	Conjunto de atributos
Atributo multivalorado	Relação + **uma chave** estrangeira
Relacionamento *n*-ário	Relação + ***n*-chaves** estrangeiras
Chave	Chave **primário** ou **alternativa**
Entidade fraca	Relação + chaves estrangeiras

Figura 3.19
Quadro Resumo com as regras de conversão ER > Modelo Lógico.

A figura 3.19 traz um quadro resumo com as regras de conversão do DER para o modelo lógico. Já a figura 3.20 mostra o resultado da conversão do caso EMPRESA, que vem sendo utilizado ao longo deste curso.



Estudo de Caso EMPRESA
DER

Exemplo:
EMPREGADO (@**SSN**, pNome, inicialM, uNome, sexoEmp, endereço, salário, dataNascEmp, **NúmeroDepto**)
SUPERVISÃO (@**SSNsupervisor**, @**SSNsupervisonado**)
DEPENDENTE (@**SSN**, @**Nome**, SexoDep, dataNascDep, parentesco)
DEPARTAMENTO (@**NúmeroDepto**, nomeDepto, numEmp, SSNChefe, dataInício)
LOCALIZAÇÃO (@**númeroDepto**, @**localização**)
PROJETO (@**NumProj**, NomeProj, Localização)
TRABALHA_EM (@**SSN**, @**NumProj**, Horas)

Modelo Lógico EMPRESA

Figura 3.20
Conversão do caso EMPRESA.

Finalmente, na figura 3.21, apresentamos o modelo lógico do estudo de caso EMPRESA, conforme este é representado na ferramenta brModelo. Repare que as chaves são indicadas através de um símbolo específico (🗝) e que cada atributo tem seu tipo definido (em alto nível).

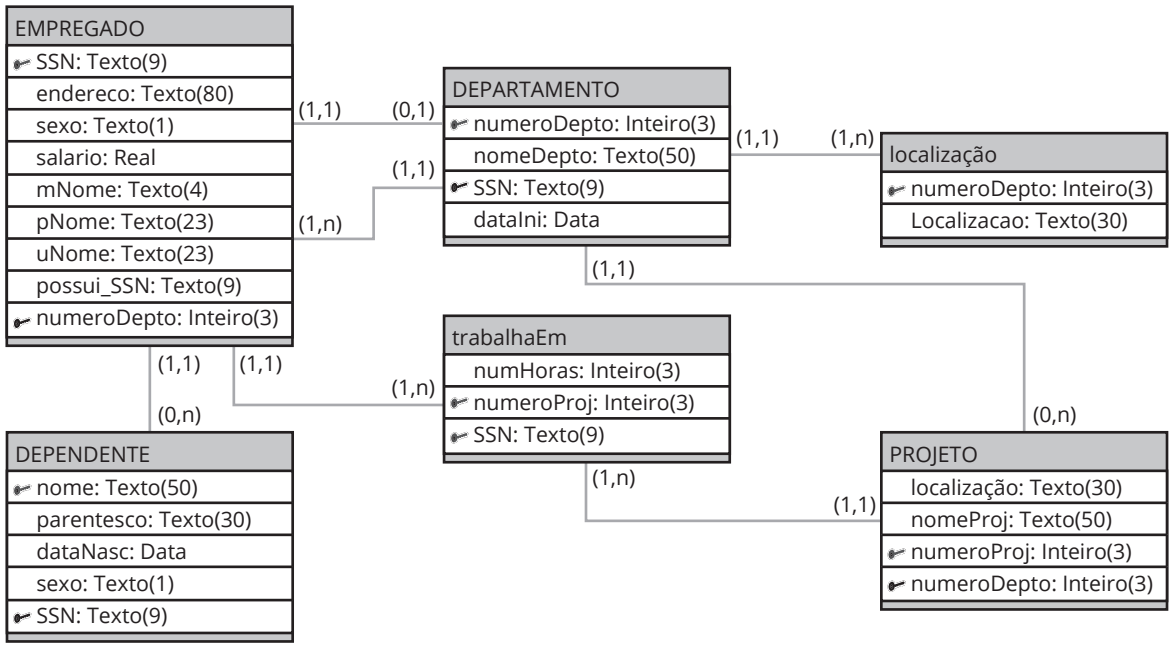


Figura 3.21
Modelo Lógico do estudo de caso Empresa no brModelo.

Exercícios de Fixação
Comandos DDL para criação do Banco de Dados

Para você, qual é a importância da criação do modelo conceitual e lógico como base para a criação do modelo físico e posterior scritp SQL/DDL a ser utilizado na criação do banco de dados da aplicação em desenvolvimento?

4

Modelo Físico

objetivos

Apresentar a Structured Query Language (SQL) como ferramenta para a criação e manutenção de modelos físicos de bancos de dados, bem como o SGBD PostgreSQL.

conceitos

Modelo físico; SQL; DDL; DML; DCL; DTL; Views; Triggers e Stored Procedures.

O modelo físico

Podemos fazer as seguintes afirmações sobre o modelo físico:

- Usa como base o modelo lógico.
- Detalhes sobre armazenamento interno dos dados.
  - Questões que influenciam o desempenho das aplicações.
- Usado por quem faz a sintonia do banco.
  - Ajuste de desempenho – "tuning".
- Auxilia na escolha do SGBDR considerando, por exemplo:
  - Tempo de resposta, utilização de espaço em disco, taxa de processamento de transações (throughput), custo etc.
- Resultado: definição de estruturas para construção do BD (script em SQL/DDL).

Ferramentas CASE podem converter o modelo lógico em um script SQL/DDL

A partir deste momento, vamos passar a nos preocupar com a criação de estruturas de dados físicas no SGBD(R), responsáveis pelo armazenamento dos dados modelados nas fases anteriores (modelos conceitual e lógico).

Estamos falando do modelo físico do banco de dados, que contém detalhes sobre como os dados serão armazenados internamente. O modelo físico vai se preocupar com questões que não têm influência direta na programação de aplicações utilizando o SGBD, mas que influenciam o desempenho destas. É utilizado por profissionais que fazem a sintonia fina do banco (ajuste de desempenho – "tuning"). Outra função do modelo físico é ajudar a escolher o SGBDR ideal para a aplicação, uma vez que serão consideradas as opções ou características que oferecem um melhor casamento entre o modelo existente e os recursos oferecidos por diferentes SGBDRs.

Como exemplos das características que são consideradas, temos: Tempo de resposta, utilização de espaço em disco, taxa de processamento de transações (throughput), custo etc.

O modelo físico é a etapa final antes de se partir para a criação do banco de dados propriamente dito, e pode ser entendido como um refinamento do modelo lógico abordado na sessão anterior. Um modelo bem estruturado e documentado é a base para a definição das estruturas necessárias para a construção do banco de dados.

É importante ressaltar que muitas das ferramentas de modelagem disponíveis no mercado possibilitam a geração do modelo físico a partir do modelo lógico desenvolvido. Isso é feito através da geração de scritps em uma linguagem denominada SQL/DDL e que será estudada com mais detalhe logo a seguir.

Na verdade, considerando todas as etapas do processo de modelagem, temos:

minimundo ⇨ DER ⇨ modelo lógico ⇨ script SQL/DDL.

A ferramenta brModelo, que temos utilizado ao longo deste curso, também é capaz de apoiar o processo de conversão do modelo lógico para o modelo físico. A figura 4.1 mostra o resultado desse processo para o estudo de caso EMPRESA.

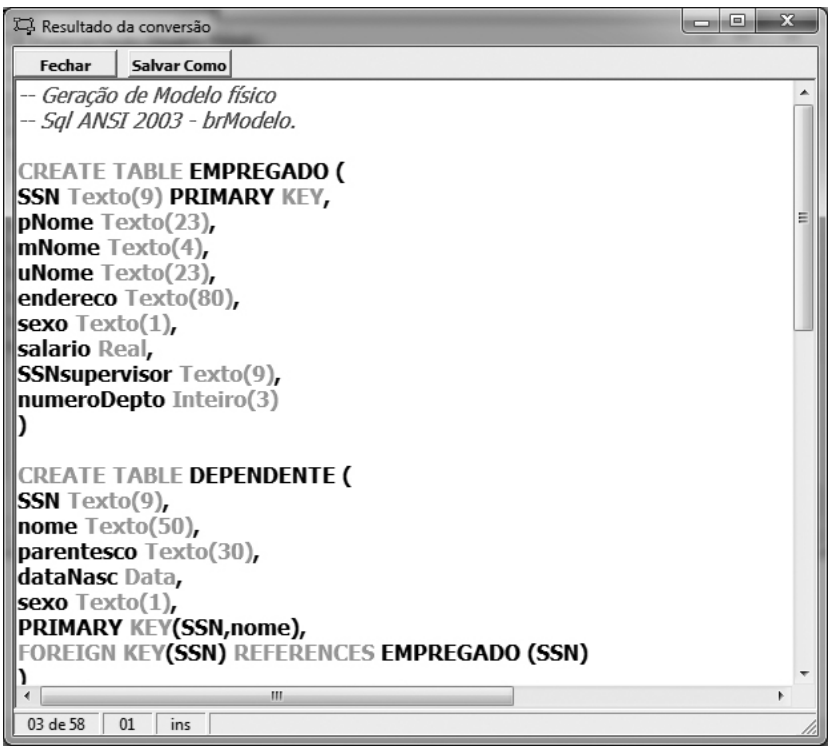


Figura 4.1
Resultado da conversão do modelo lógico para físico no brModelo.

O resultado apresentado na figura 4.1 é uma representação bem aproximada da definição das estruturas de dados a serem implementadas fisicamente no SGBDR. No caso do brModelo, os tipos de dados dos domínios de cada atributo são gerados com informações de mais alto nível, já que essa ferramenta não se preocupa com os tipos de dados disponíveis nos diferentes SGBDs disponíveis no mercado.

Assim, será necessário um passo adicional para usar o resultado gerado pelo brModelo na criação propriamente dita das estruturas de dados (tabelas, RI etc.). Qual seja: mapear os tipos de dados definidos no brModelo para cada atributo de cada tabela nos respectivos tipos de dados disponíveis no SGBD a ser utilizado.

Structured Query Language (SQL)

A Structured Query Language (SQL) é a linguagem padrão para trabalhar com banco de dados relacionais nos diferentes SGBDs disponíveis no mercado.

A primeira versão, originalmente chamada de Structured English Query Language (Sequel), foi desenvolvida pela IBM no início da década de 70, demonstrando a viabilidade da implementação do modelo relacional proposto por Codd. Desde então, a linguagem evoluiu bastante, passando a se chamar simplesmente Structured Query Language (SQL) e se estabeleceu claramente como a linguagem padrão utilizada em banco de dados relacionais.

Em 1986, o American National Standards Institute (ANSI) e a International Organization for Standardization (ISO) publicaram o primeiro padrão SQL. De lá para cá novas versões, com sucessivos aperfeiçoamentos foram publicadas:

- SQL-86;
- SQL-89: publicação de um padrão estendido para linguagem em 1989;
- SQL-92;
- SQL-1999;
- SQL-2003 (XML);
- SQL-2008;
- SQL-2011 ISO/IEC 9075:2011 (16/12/2011).

Saiba mais

Cabe ressaltar, contudo, que apesar da maior parte dos SGBDs ser compatível com uma das versões do padrão, não são raros os que acabam desenvolvendo extensões ou variações em relação ao padrão adotado.

A SQL usa uma combinação de construtores em Álgebra e Cálculo Relacional (tuplas), cujo entendimento é imprescindível para quem quiser se aprofundar no uso de bancos de dados relacionais. Uma referência importante de consulta sobre o assunto é o livro *Sistemas de banco de dados*, de Ramez Elmasri e Shamkant B. Navate.

Estrutura da SQL

Na verdade, a SQL é composta por um conjunto de linguagens que permitem a definição, consulta e atualização de dados, além de recursos adicionais para a segurança e gestão do BD. São elas:

- **DDL**: Linguagem de Definição de Dados;
- **DML**: Linguagem de Manipulação de Dados;
- **DCL**: Linguagem de Controle de Dados;
- **DTL**: Linguagem de Controle de Transações.

O conjunto de comandos da linguagem DDL é usado para a definição das estruturas de dados, fornecendo as instruções que permitem a criação, modificação e remoção de objetos de banco de dados. A DDL trabalha com os metadados (dados acerca dos dados) que ficam armazenados no dicionário de dados (catálogo). Exemplos de comandos dessa linguagem são: CREATE, ALTER e DROP.

DML é o grupo de comandos dentro da linguagem SQL utilizado para a recuperação, inclusão, remoção e modificação de informações no banco de dados. Aqui são realizadas as operações CRUD (Create, Retrieve, Update e Delete), através de comandos tais como: *INSERT*, *SELECT*, *UPDATE* e *DELETE*.

DCL é o grupo de comandos que permitem ao administrador de banco de dados gerenciar os aspectos de autorização de dados e licenças de usuários para controlar quem tem acesso para ver ou manipular dados dentro do banco de dados. Exemplos de comandos da DCL são: *GRANT* e *REVOKE*.

DTL, por sua vez, é a linguagem de Controle de Transações e fornece mecanismos para controlar transações no banco de dados. Seus comandos mais conhecidos são: *BEGIN TRANSACTION*, *COMMIT* e *ROLLBACK*.

A DDL será abordada com mais detalhes ainda nesta sessão. A DML, por sua vez, será objeto das sessões seguintes, enquanto que a DCL não faz parte do escopo desse curso (ela é estudada no curso que trata da Administração de Bancos de Dados).

Saiba mais

A DTL será discutida na sessão 10.

Outras funcionalidades

Existem algumas funcionalidades gerais, fornecidas pelos principais SGBDs disponíveis no mercado, que servem para facilitar, proteger e automatizar alguns dos processos realizados dentro do banco de dados.

Entre elas, destacamos:

- Views;
- Triggers e Stored Procedures;
  - MySQL-SQL/PSM, Oracle-PL/SQL, PostgreSQL-PL/pgSQL/PSM.
- "Regras" para embutir comandos SQL em Linguagens de programação.

Views são utilizadas para facilitar o acesso a um determinado conjunto de dados. Esse recurso possibilita a criação de seleções de dados, provenientes de tabelas, mas disponibilizados por meio de outras tabelas que atuam como filtros. Esses filtros, na verdade, são uma restrição de dados, baseados em regras estipuladas pelo usuário, para que somente os resultados desejados sejam disponibilizados, e não todos os campos e registros de uma determinada tabela do banco de dados.

Os triggers, geralmente escritos em DML, não são disparados pelos usuários, mas sim pela ativação de algum evento ocorrido no banco de dados.

As Stored Procedures (procedimentos armazenados) são uma coleção de comandos em SQL pensados e implementados com o objetivo de aperfeiçoar consulta no banco de dados. Por exemplo, os procedimentos armazenados podem ser utilizados para reduzir o tráfego na rede, melhorar a performance do banco de dados, bem como criar mecanismos de segurança, entre outras funcionalidades.

Por fim, as empresas que disponibilizam SGBDs no mercado também se preocupam em desenvolver e disponibilizar regras para o uso de seus produtos nas diversas linguagens de programação em uso, visando manter um padrão para facilitar o processo de como seus comandos poderão ser embutidos nas soluções de software em desenvolvimento que manipulam banco de dados.

PostgreSQL

- Origem no projeto Postgres, na Universidade de Berkeley, em 1986.
- Em 1996, recebeu o nome atual PostgreSQL.
- Licença de uso BSD, open source.
- Apresenta principais recursos existentes nos bancos de dados pagos disponíveis no mercado.

- Compatível com diversos SOs.

Possui bibliotecas e extensões para as principais plataformas e linguagens de programação.

Neste curso, estaremos aplicando os conceitos de banco de dados no PostgreSQL, que é um sistema de gerenciamento de banco de dados relacional (SGBDR), utilizado para gerenciar dados armazenados em relações entre tabelas.

Relação é, essencialmente, um termo matemático para tabela. A noção de armazenar dados em tabelas é tão trivial hoje em dia que pode parecer totalmente óbvio, mas existem várias outras formas de organizar bancos de dados. Arquivos e diretórios em Sistemas Operacionais tipo Unix são um exemplo de banco de dados hierárquico. Um desenvolvimento mais moderno são os bancos de dados orientados a objeto.

Cada tabela é uma coleção nomeada de linhas. Todas as linhas de uma determinada tabela possuem o mesmo conjunto de colunas nomeadas, e cada coluna é de um tipo de dado específico. Enquanto as colunas possuem uma ordem fixa nas linhas, é importante lembrar que o SQL não garante a ordem das linhas dentro de uma tabela (embora as linhas possam ser explicitamente ordenadas para a exibição).

As tabelas são agrupadas em bancos de dados, e uma coleção de bancos de dados gerenciados por uma única instância do servidor PostgreSQL forma um agrupamento de bancos de dados.

Características do PostgreSQL

Limit	Value
Maximum database size	Unlimited
Maximum table size	32 TB
Maximum row size	1.6 TB
Maximum field size	1 GB
Maximum rows per table	Unlimited
Maximum columns per table	250-1600 depending on column types
Maximum indexes per table	Unlimited

Figura 4.2
Características do PostgreSQL.

Outras informações relevantes são:

- Download do PostgreSQL (e documentação).
  - http://www.postgresql.org
- Instalação:
  - Porta: 5432 e Usuário: postgres, senha.
  - Estrutura de pastas.
  - Arquivo de configuração.
  - Incluso cliente (front-end) pgAdmin3.
- Arquivo de configuração: PostgreSQL\9.x\data\postgresql.conf
- Diretório padrão para os dados: PostgreSQL\9.x\data\base

Criação do banco de dados

Temos de partir do princípio de que temos um SGBD instalado e que seus serviços estejam disponíveis para que os modelos físicos de banco de dados em desenvolvimento possam ser criados.

- Necessário ter iniciado o SGBD PostgreSQL.
  - Instalado como serviço (início automático ou não).

Figura 4.3
Tela que disponibiliza serviços do SO.

Neste curso, vamos utilizar o PostgreSQL como SGBD, sendo necessário que ele esteja configurado como um serviço ativo para possibilitar acesso à criação de banco de dados e suas estruturas de dados.

O próximo passo que devemos ter em mente é realização de uma conexão com o SGBD. Existem diversas ferramentas disponíveis no mercado, sendo o psql e pgAdmin3 duas ferramentas que acompanham a versão do postgreSQL disponibilizada para este curso.

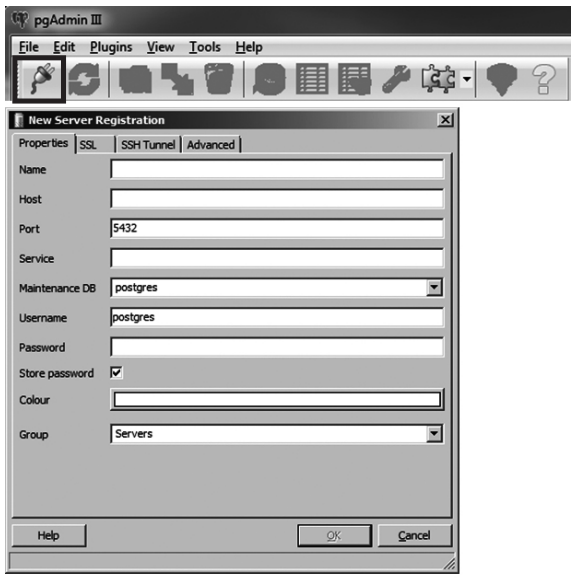
- Comandos SQL podem ser executados:
  - Via console (modo texto): psql
  - Via ferramenta gráfica: pgAdmin3
  - Ou embutidos em linguagens de programação.
- Observações:
  - SQL não é case-sensitive.
  - Comandos terminam com ponto-e-vírgula (;).

Criar conexão com o SGBD postgreSQL (no pgAdmin3)

Neste ponto, algumas informações são indispensáveis. As informações são em parte fornecidas durante o processo de instalação do SGBD postgreSQL, sendo elas o endereço (IP ou DNS) e porta de comunicação que estará aguardando requisições de conexão para manipulação de banco de dados.

As informações complementares são o usuário e senha, para que seja possível a disponibilização de acesso. Inicialmente temos um usuário, conhecido como super-usuário na documentação oficial do postgreSQL, que possui todos os privilégios necessários configurados para manipulação do banco de dados no SGBD postgreSQL. Neste curso, estaremos utilizando esse super-usuário criado durante o processo de instalação do postgreSQL, o postgres.

- Criar conexão com o servidor PostgreSQL no pgAdmin3



Informar endereço e porta do servidor e também usuário e senha

Note que superusuários tem todos privilégios automaticamente

Figura 4.4
pgAdmin3:
Tela de conexão
com o SGBD.

A criação de uma conexão ainda não dá acesso a um banco de dados específico, e sim nos disponibiliza uma ponte com informações necessárias para abrir uma conexão com o banco de dados que desejamos trabalhar.

Para executar qualquer comando, é necessário conectar ao servidor PostgreSQL (no pgAdmin3).

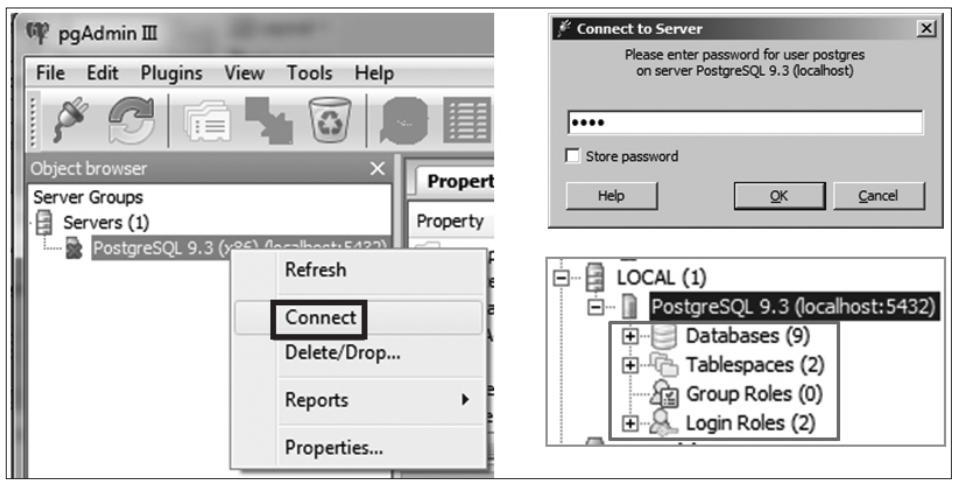


Figura 4.5
pgAdmin3 – Passos
para ativar conexão
ao servidor
postgreSQL.

Comando SQL/DML para criar banco de dados

O conjunto de comandos da linguagem DDL é usado para a definição das estruturas de dados, fornecendo as instruções que permitem a criação, modificação e remoção de objetos de banco de dados (base de dados, esquemas, tabelas, índices etc.).

Banco de dados:

- Objeto básico em um SGBD que contém todos os objetos que serão criados para esse banco de dados.
- Usuário com direito de acesso CREATEDB.
  - CREATE DATABASE <nomeDB> [argumentos];
- Argumentos:
  - OWNER usuário (usuário com amplos poderes sobre o DB);
  - TEMPLATE nomet (modelo padrão para a DB);
  - ENCODING valor (conjunto de caracteres: LATIN1, UTF8...);
  - TABLESPACE nomets (local de armazenamento do DB);
  - CONNECTION LIMIT (número de conexões simultâneas).

Tablespaces são definições de locais para armazenamento lógico das informações do servidor. É visto como diretórios existentes em seu SO. Sua funcionalidade é a de possibilitar o armazenamento de informações do servidor em locais distintos, sendo motivado por políticas de backup, utilização de mais de um HD, organização, entre outros.

- Tablespaces são definições de locais de armazenamento lógico dos dados (por questão de organização, política de backup, utilização de mais de um disco rígido etc.).
- Trata-se de diretórios no SO.
- É possível gerenciar tablespace (criar, alterar e excluir):
  - CREATE TABLESPACE <nomeTS> LOCATION 'local';
- Connection Limit por padrão é -1 e representa sem limite (variável max_connections no arquivo de configuração do PostgreSQL).

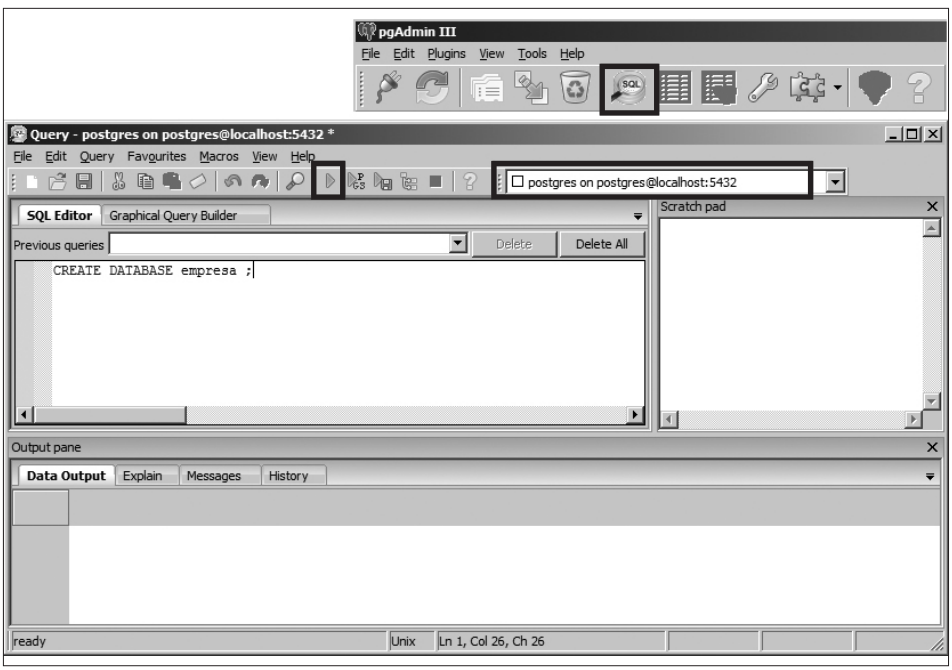


Figura 4.6
pgAdmin3 – Uso do SQL Editor para criação de banco de dados no postgreSQL.

A ferramenta SQL Editor, parte integrante do pgAdmin3, disponibiliza uma interface onde podemos digitar as instruções SQL a serem utilizadas para a criação do banco de dados.

Outro recurso que podemos usar é o assistente para a criação de banco de dados oferecido pelo pgAdmin3 (figura 4.7).

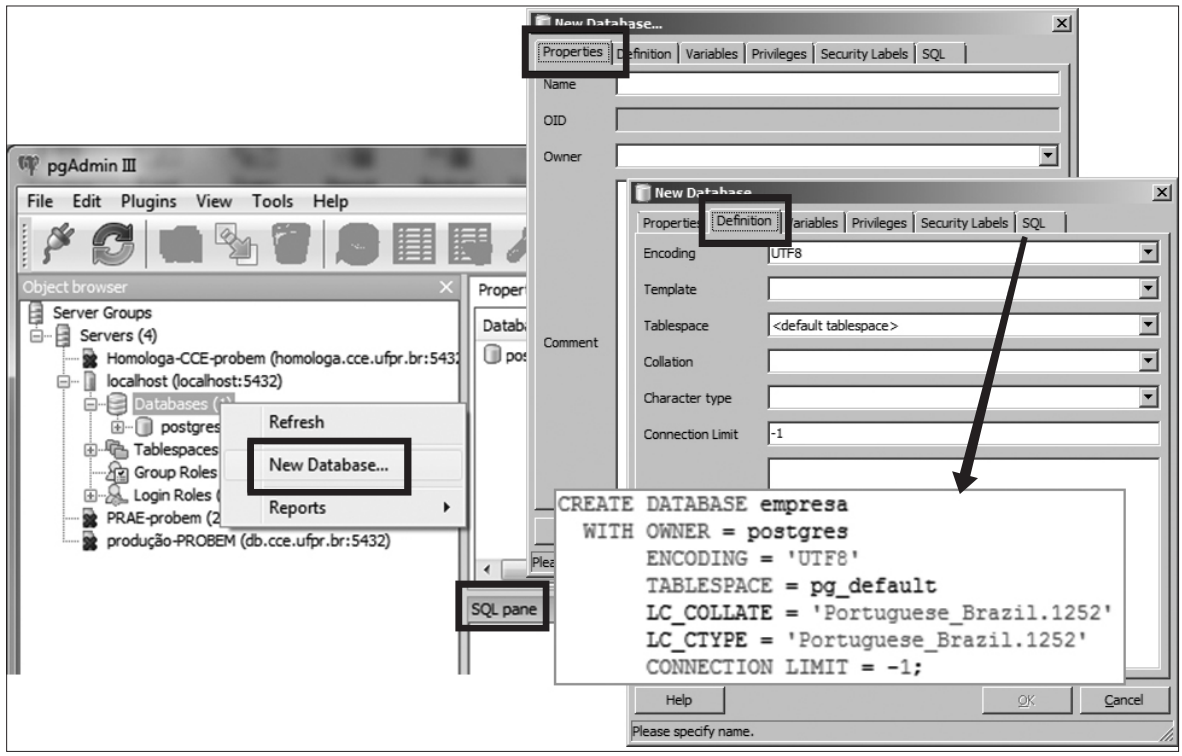


Figura 4.7
pgAdmin3 – Uso de assistente para criação de banco de dados no postgreSQL.

A conexão com o banco de dados só será possível após a criação e posterior conexão com o este. Apenas após a confirmação de conexão estabelecida é que poderemos dar início ao processo de execução de consultas no banco de dados.

Outro ponto importante a se ressaltar é que o acesso aos objetos de um determinado banco de dados também só estará disponível para uso após a confirmação da conexão (figura 4.8).

Para executar comandos sobre um banco de dados específico, é necessário conectar-se a ele: clique sobre o nome do BD.

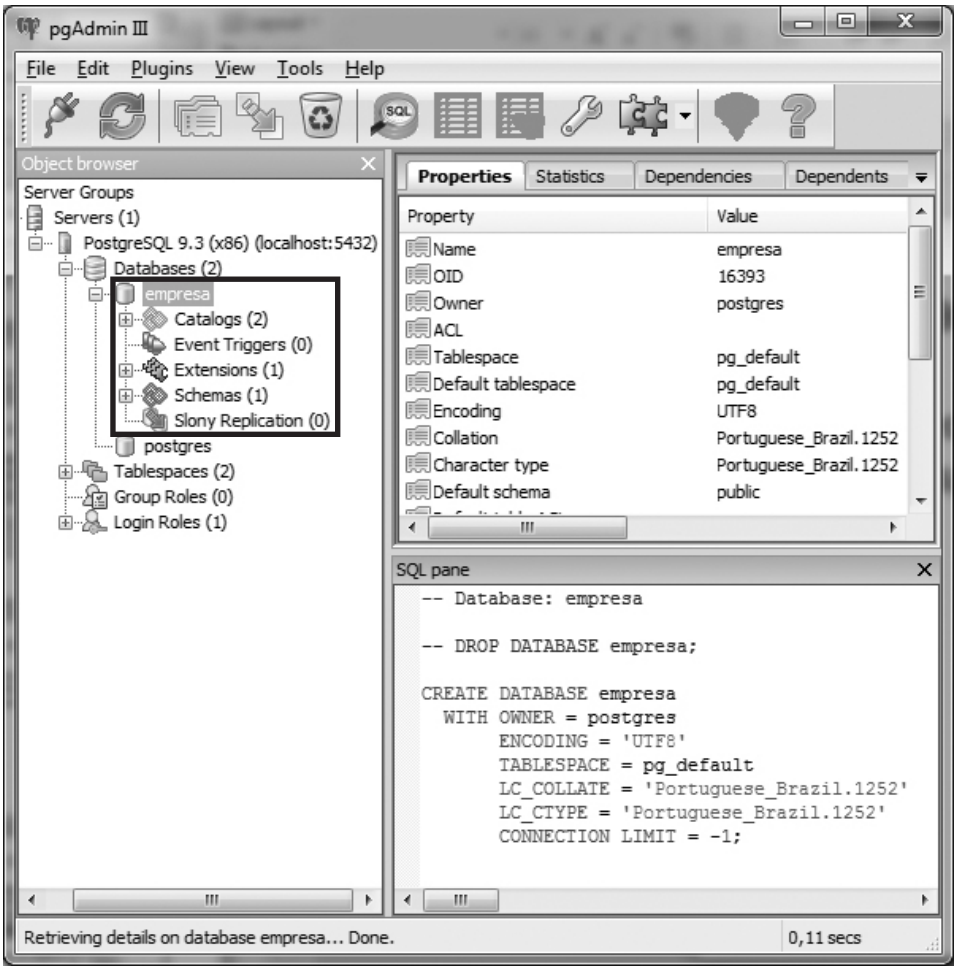


Figura 4.8
pgAdmin3 – Acesso à estrutura do banco de dados após ativação de conexão.

Alteração do banco de dados

Uma vez criado o banco de dados, é possível procedermos com a alteração de seu nome, bem como a alteração de seu proprietário. Essa operação pode ser realizada via instrução SQL no utilitário SQL Editor ou por meio da edição das propriedades do banco de dados pelo pgAdmin3 (figura 4.9).

```
ALTER DATABASE <nomeDB> RENAME <novoNomeDB>;

ALTER DATABASE <nomeDB> TO OWNER <novoUser>;
```

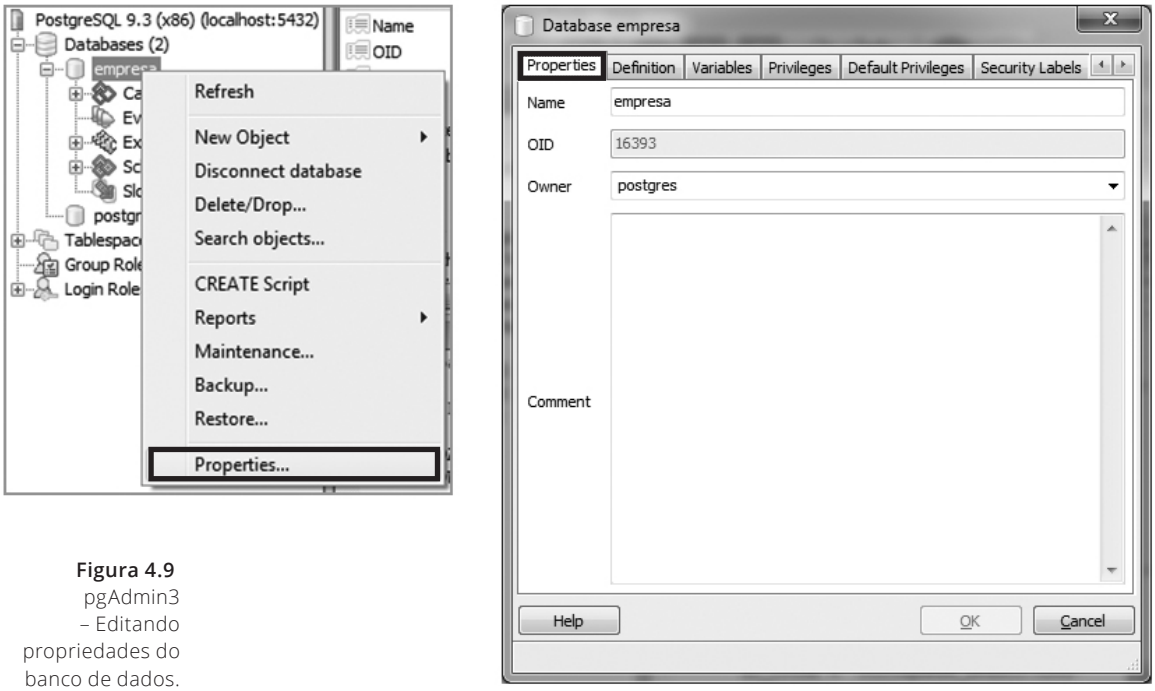


Figura 4.9
pgAdmin3 – Editando propriedades do banco de dados.

Exclusão do banco de dados

Por fim, podemos proceder com a execução de um determinado banco de dados, podendo ser realizada via instrução SQL no utilitário SQL Editor ou por meio da edição das propriedades do banco de dados pelo pgAdmin3 (figura 4.10).

EXCLUSÃO: Apaga todas as tabelas e estruturas associadas e, consequentemente, todos os dados existentes.

```
DROP DATABASE <nomeDB>;
```

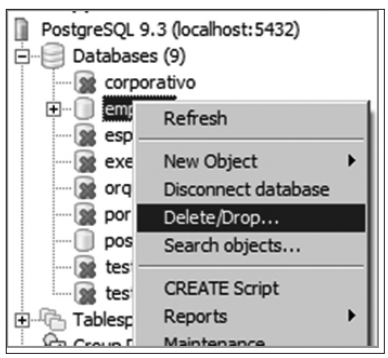


Figura 4.10
pgAdmin3 – Excluindo um banco de dados.

Esquema de banco de dados

Esquemas podem ser entendidos como pastas utilizadas de objetos criados em um banco de dados. Um banco de dados pode ter dezenas de objetos, como tabelas, visões e índice, entre outras estruturas. Esquemas possibilitam a organização, como por exemplo, agrupados de acordo com seu uso.

Recomendamos o uso de esquemas em um banco de dados quando for necessário o agrupamento e organização de objetos que posteriormente terão o acesso disponibilizado para um determinado usuário ou grupo de usuários.

ESQUEMA:

- Agrupa ou organiza tabelas, restrições, visões, domínios e outros objetos (construtores) do BD;
- Incorporado no SQL2: inicialmente todas as tabelas eram de um mesmo esquema;
- Na prática é o “banco de dados”;
- Esquema padrão: 'public'.

```
CREATE SCHEMA <nomeSh> [argumentos];
```

- Argumentos:
  - AUTHORIZATION usuário

A figura 4.11 tem informações sobre a alteração de Esquemas.

Alteração do Esquema: modifica algumas propriedades

ALTER SCHEMA <nomeSh> RENAME TO <novoNomeSh>;

ALTER SCHEMA <nomeSh> OWNER TO <novoUser>;

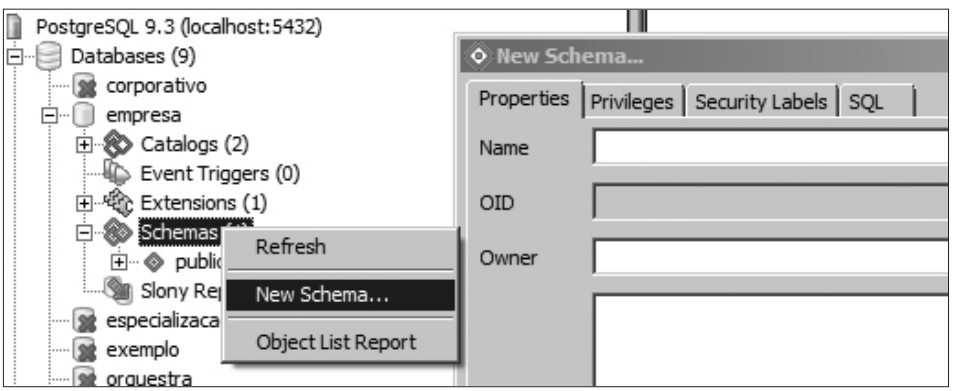


Figura 4.11
pgAdmin3 – Criando esquema em um banco de dados.

A exclusão de esquemas de um banco de dados também está disponível para ser realizada por linha de comando SQL no SQL Editor ou por meio da interface gráfica do pgAdmin3 (figura 4.12).

Exclusão do Esquema e de todas as estruturas que ele contém.

```
DROP SCHEMA <nomeSh> [CASCADE|RESTRICT];
```

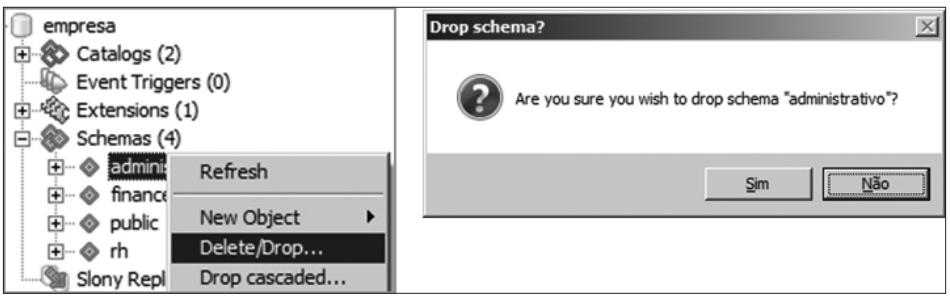


Figura 4.12
pgAdmin3 – Excluindo esquema em um banco de dados.

Criando tabelas no banco de dados

O gerenciamento estrutural de tabelas é um dos conjuntos de comandos mais completos e customizáveis dentro da estrutura de um SGBD. Esse conjunto de comandos possibilita desde a criação de bancos de dados, de tabelas, assunto desse tópico, índices, visões, entre outros.

TABELA:

- Estrutura do BD derivada de uma relação do modelo lógico.
  - Possui campos (mapeados dos atributos da relação);
  - Define o domínio dos campos: tipo de dado;
  - Define o conjunto de restrições de integridade (RI).

```
CREATE TABLE <nomeTb>(
   <nomeCampo1> <tipoDado> [<restrições>],
   <nomeCampo2> <tipoDado> [<restrições>],
   ... [<restriçõesTabela>]
);
CREATE TABLE <nomeSh>.<nomeTb>(...);
```

A figura 4.13 apresenta a interface para criação de tabelas através do assistente do PgAdmin3.

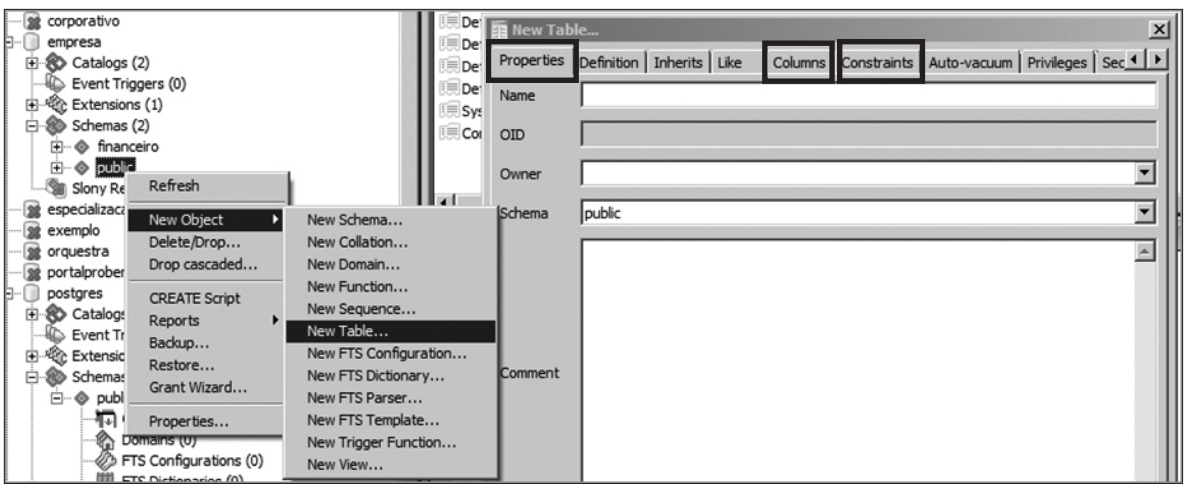


Figura 4.13 pgAdmin3 – Criando tabelas.

Existe uma grande variedade de tipos de dados que podem ser utilizados no processo de construção de campos que compõem uma tabela. Neste curso, vamos listar os mais comumente utilizadas por proximidade com a realidade das informações geralmente manipuladas nos bancos de dados. São eles:

Principais tipos de dados

- **Numérico**: bigint (int8), integer (int, int4), smallint (int2), serial, double precision (float8), real (float4), numeric [(p,s)] (decimal), money;
- **Literal**: character [(n)] (char), character varying[(n)] (varchar), text, bit varying[(n)] (varbit);
- **Lógico**: boolean(bool);
- **Temporal**: date, time, timestamp [with(out) time zone];
- **Outros**: bytea, oid, tipos enumerados, tipos geométricos, tipos endereços de rede...

A figura 4.14 exibe as janelas relacionadas com a criação dos campos de uma tabela usando o assistente do PgAdmin3.

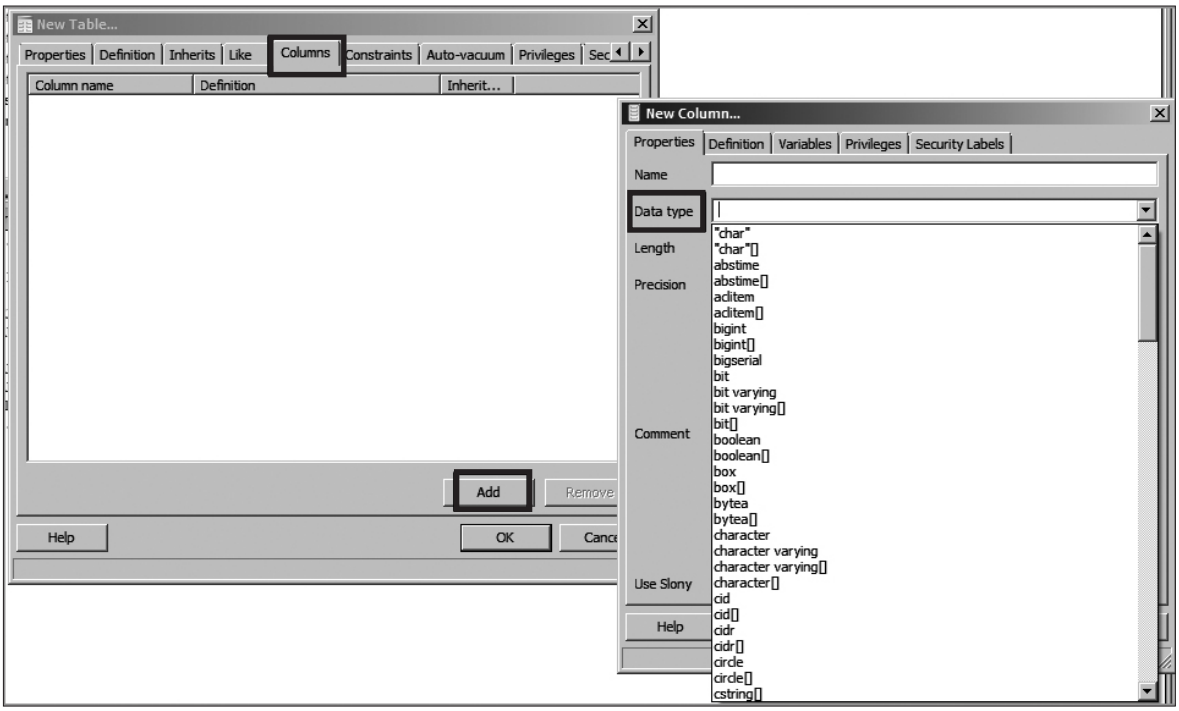


Figura 4.14
pgAdmin3 – Assistente para criação de colunas em uma tabela.

Tipos de dados, que compõem os campos ou colunas de uma tabela no banco de dados, poderão ter de atender a determinadas restrições, que são derivações de regras do mini-mundo que o banco de dados representa. Geralmente temos um conjunto considerável de restrições para os valores reais sendo manipulados. Em determinada situação essas restrições poderão ser tratadas na camada de visão da aplicação que manipula os dados, mas, porém, em algumas situações, esses deverão ser tratadas na estrutura das tabelas pertencentes a ao banco de dados em uso.

Restrições (constraints):

- São regras a que os valores de uma ou mais colunas ou campos devem obedecer;
- A utilização de constraints é a única garantia de que os dados existentes nas colunas estão de acordo com as regras especificadas no projeto do banco de dados;
- Garante correção nos dados, pois erros de programação podem fazer com que sejam aceitos;
- Podem ser restrições de coluna ou de tabela.

Restrições de atributos tratam especificamente as restrições de chave e referencial, bem como as restrições de domínio de atributos.

Restrições em Atributo:

- Se nada for indicado de restrição sobre um atributo, por default, ele admite o valor nulo (NULL):
  - Representa um atributo opcional.
- Se não for um atributo opcional, então acrescentar a cláusula NOT NULL;
- É possível associar um valor padrão (que não nulo) através da cláusula DEFAULT <valor>.

O pgAdmin3 disponibiliza recurso visual gráfico para auxiliar no processo de definição de restrição de atributos durante a sua criação.

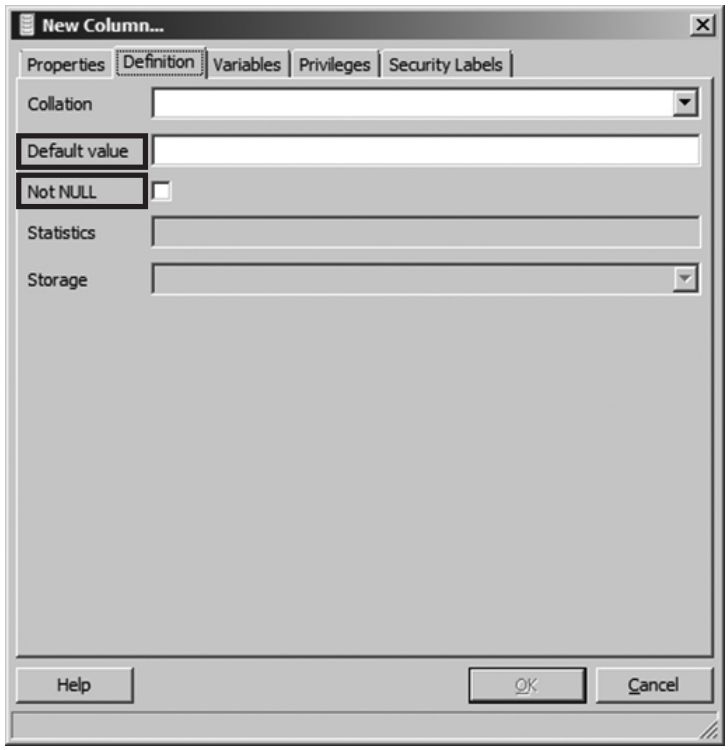


Figura 4.15
pgAdmin3 – Assistente para definição de restrições em colunas ou campos.

Restrições de chave e de integridade referencial são muito importantes. Por isso existem cláusulas especiais dentro do comando *CREATE TABLE* para especificá-las.

- Cláusula UNIQUE: Permite indicar que os valores da coluna não podem repetir.
  - Pode ser restrição de tabela: UNIQUE <campo[s]>
  - Representa atributo único.
- Restrição de Chave Primária.
  - Indicação do atributo identificador da tabela (se existir): uso da cláusula PRIMARY KEY.
  - Em uma tabela, só pode existir apenas uma cláusula PRIMARY KEY. Se a chave primária for concatenada, utilizar restrição de tabela: PRIMARY KEY <campo[s]>
  - É uma combinação de NOT NULL + UNIQUE.
- Restrição de Integridade Referencial.
  - Permite inserir atributos relacionantes (chave estrangeira) em uma tabela: uso da cláusula REFERENCES <TabelaRef> (<campoRelacionante>);
  - O valor campo será validado em sua origem;
  - Na restrição de tabela: FOREIGN KEY <campo>
  - REFERENCES <TabelaRef> (<campoRelacionante>).

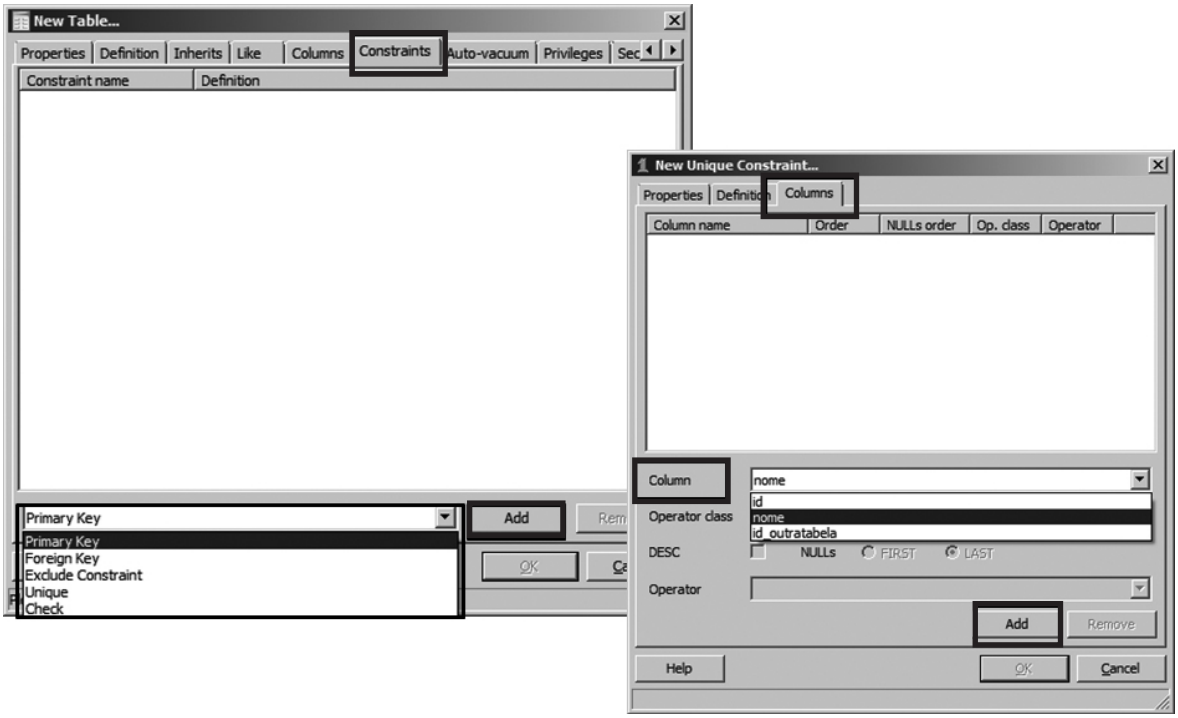


Figura 4.16
pgAdmin3 – Assistente para definição de restrições em atributos.

Importante:

- Por padrão, se nenhum nome for atribuído à restrição UNIQUE, receberá o nome nomeTB_nomeCOL_key;
- Por padrão, se nenhum nome for atribuído à restrição PRIMARY KEY, receberá o nome nomeTB_pkey;
- Por padrão, se nenhum nome for atribuído à restrição FOREIGN KEY, receberá o nome nomeTB_nomeCOL_fkey;

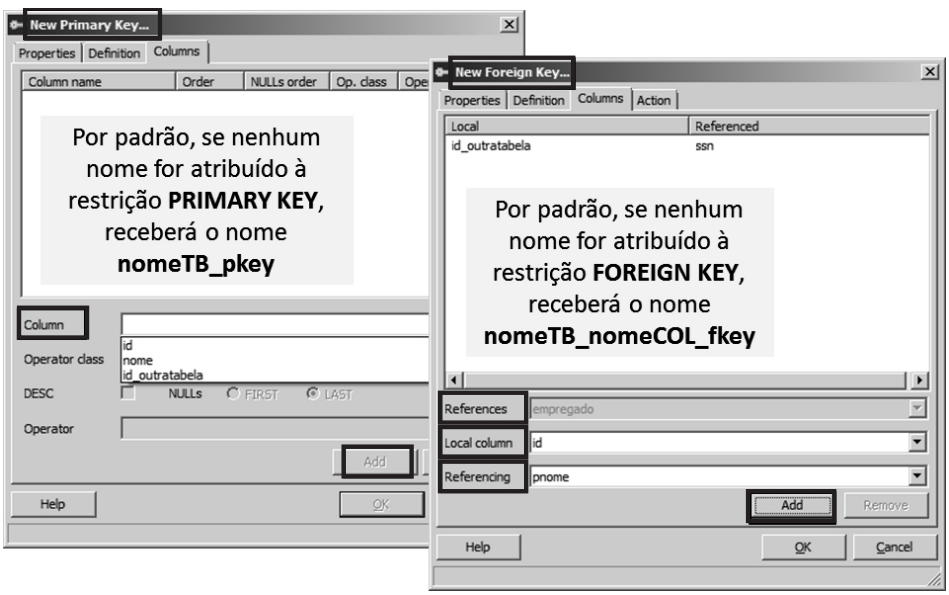


Figura 4.17
pgAdmin3 – Assistente para definição de chave primária e integridade referencial.

Violação da Integridade Referencial:

- Ocorre quando tuplas são inseridas, excluídas ou valor do atributo chave (primária ou estrangeira) é modificado;
- Ação padrão da SQL: rejeitar;
- Projetista pode especificar ação referencial engatilhada (referential triggered action), integrada com ON DELETE ou ON UPDATE.
  - SET NULL ou SET DEFAULT ou CASCADE.
- Como as operações de exclusão e atualização no lado 1 refletirão no lado N.

Exemplo:

- Se a tupla de um empregado supervisor for excluída, o valor de codEmpSup será marcado como NULL em todas as tuplas de empregado que fizer referência ao Supervisor excluído;
- Se a tupla de um empregado supervisor for atualizada, o novo valor será propagado em todas as tuplas de empregado que fizer referência ao Supervisor atualizado.

```
CREATE TABLE empregado (
 codEmp INTEGER PRIMARY KEY,
 codEmpSup INTEGER REFERENCES EMPREGADO(codEmp)
   ON DELETE SET NULL
   ON UPDATE CASCADE,
...);
```

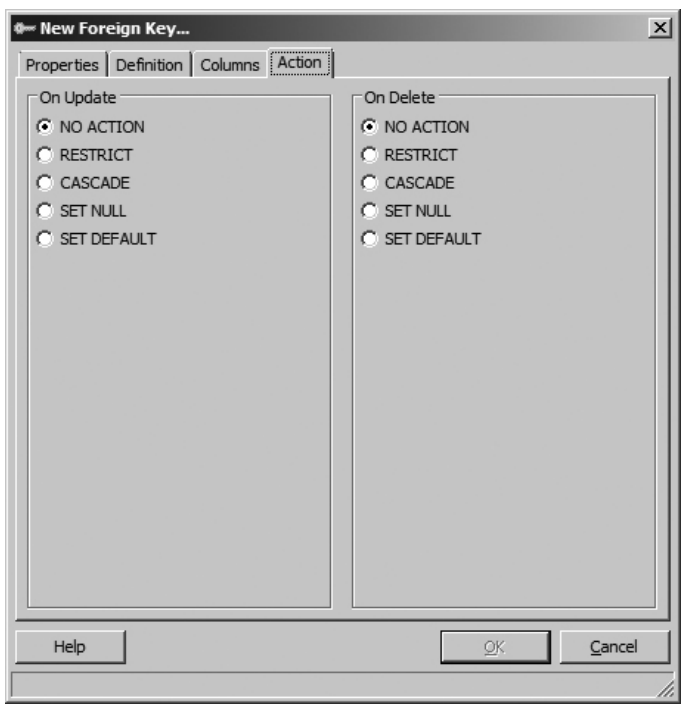


Figura 4.18 pgAdmin3 – Assistente para criação de restrições.

Restrições em Atributo:

- É possível validar dados inseridos em um campo através da cláusula CHECK(<condição>);
- É a especificação de uma condição, cujo resultado da avaliação deve ser verdadeiro para que os dados sejam aceitos (limita o valor do domínio do atributo);

- Observação: restrições CHECK são úteis para garantir a integridade dos dados, mas podem representar alto custo de processamento.

Denominando as Restrições: A cláusula CONSTRAINT permite nomear uma restrição (é opcional). No pgAdmin3, é automática a atribuição de nome à restrição (conforme apresentado antes). Por padrão, se nenhum nome for atribuído à restrição check, receberá o nome nomeTB_nomeCOL_check.

```
CONSTRAINT <nomeCT> <restrição>
```

Exemplo:

```
CREATE TABLE aluno (
codaluno number(5), nome varchar2(50), dtnasc date,
mae varchar2(50), sexo char(1), curso number(5),
CONSTRAINT pk_aluno PRIMARY KEY (codaluno),
CONSTRAINT uk_aluno UNIQUE(nome, mae),
CONSTRAINT ck_sexo CHECK (sexo IN('M', 'F')),
CONSTRAINT fk_curso FOREIGN KEY (curso) REFERENCES curso(codcurso));
);
```

Alteração de tabelas no banco de dados

A alteração de uma tabela envolve um conjunto de comandos para realizar as mais diversas ações possíveis, entre as quais renomear a tabela, suas colunas, inserir ou novas colunas de dados, entre outras.

```
ALTER TABLE <nomeTB> <ações>;
```

- Ações:
  - ADD [COLUMN] <nomeCOL> <tipoDado> [<restrições>];
  - ALTER [COLUMN] <nomeCOL> **TYPE** <tipoDado>;
  - DROP [COLUMN] <nomeCOL> [RESTRICT|CASCADE];
  - RENAME COLUMN <nomeCOL> TO <novoNomeCOL>;
  - ADD CONSTRAINT <restriçõesTB>;
  - DROP CONSTRAINT <nomeCT> [RESTRICT|CASCADE];
  - OWNER TO <novoUsuário>;
  - RENAME TO <novoNomeTB>;

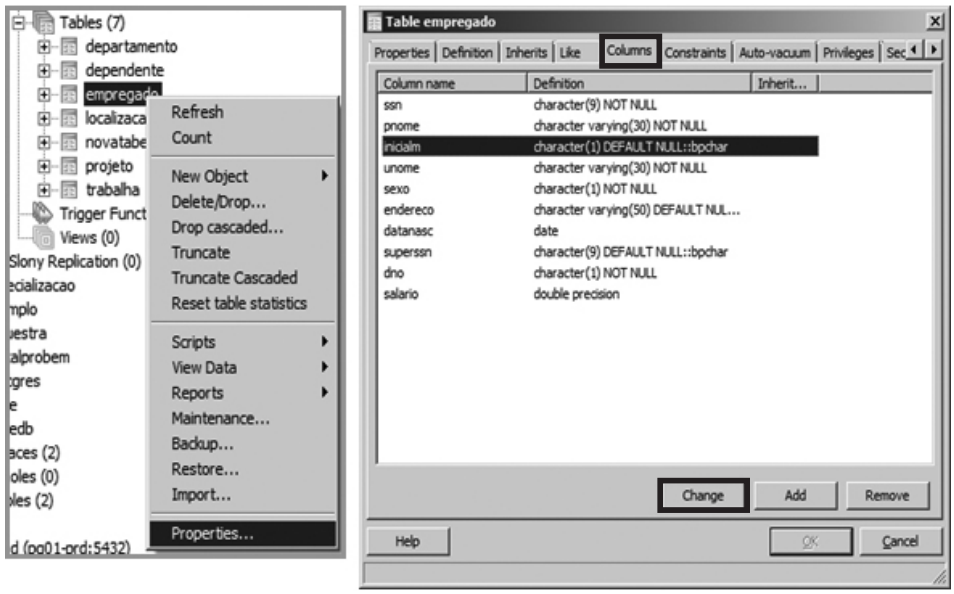


Figura 4.19
pgAdmin3 – Assistente para alteração de tabela do BD.

Observações:

- A nova coluna é inicialmente preenchida com o valor padrão (se fornecido) ou NULL;
- PostgreSQL tentará converter o valor da coluna para o novo tipo, bem como quaisquer restrições que envolvem a coluna. Mas essas conversões podem falhar;
- Muitas vezes é melhor excluir as restrições e depois adicionar novamente no novo formato.

Exclusão de tabelas no banco de dados

Para excluir uma tabela (e todos os dados lá existentes), basta utilizar o comando *DROP TABLE*, informando o nome da tabela que se deseja remover do banco de dados.

```
DROP TABLE <nomeTB> [CASCADE|RESTRICT];
```

- **RESTRICT**: se a tabela estiver referenciada em visões ou regras de integridade referencial, o comando falhará;
- **CASCADE**: sempre terá sucesso e views e constraints também serão eliminadas.

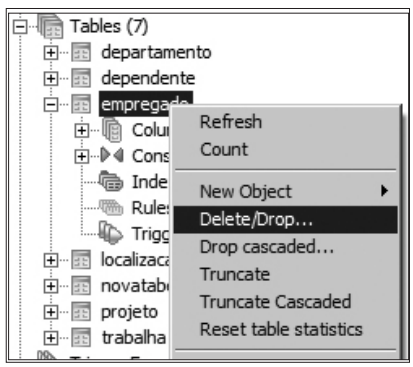


Figura 4.20
pgAdmin3 – Assistente para exclusão de tabela do BD.

Domínios

Domínios são objetos que possibilitam a criação de novos tipos de dados que poderão ser usados no processo de definição de colunas de dados dentro das tabelas sendo criadas no banco.

Domínios também são conhecidos como máscaras de dados por sua funcionalidade. Por exemplo, é possível criar um domínio de dado que valide código postal, outro que validade número telefônico, entre outros.

Domínio:

- Tipo de dado para um domínio que é utilizado para vários atributos
- Objetivo: Melhorar legibilidade do esquema
  - **CREATE DOMAIN** <nomeDM> **AS** <tipoDado>[<restrições>];
- Alteração do Domínio
  - **ALTER DOMAIN** <nomeDM> <ações>;
- Exclusão do Domínio
  - **DROP DOMAIN** <nomeDM> [CASCADE|RESTRICT].

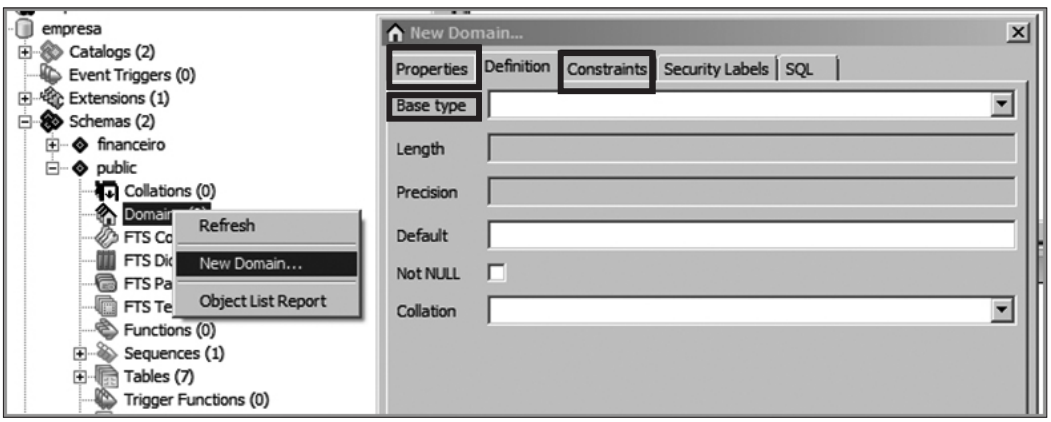


Figura 4.21
pgAdmin3 – Assistente para manutenção de domínios.

Exemplos:

```
CREATE DOMAIN tipoCEP AS CHAR(8);

CREATE DOMAIN tipoNota AS DECIMAL(3,1)

   CHECK (VALUE >=0 AND VALUE <=10);

CREATE DOMAIN dom_username TEXT

   CHECK (LENGTH(VALUE) > 3 AND

     LENGTH(VALUE) < 200 AND

     VALUE ~ '^[A-Za-z][A-Za-z0-9]+$');
```

Sequência

- É um objeto do BD que gera números em ordem sequencial.
- Aplicações, na maioria das vezes, usam esses números quando eles exigem um valor único em uma tabela, tais como valores de chave primária.

- Alguns SGBDs usam o conceito de "autoincremento" na configuração de tipos numéricos.
- Objeto criado implicitamente com um campo SERIAL.
- As sequências são manipuladas usando instruções SQL.

Diferentemente de outros banco de dados, os campos conhecidos como AUTOINCREMENT no PostgreSQL são configurados de forma separada das tabelas onde serão utilizados. Isso se deve ao fato de um campo desse tipo poder gerenciar o número de incremento de duas ou mais tabelas de forma sincronizada.

Sintaxe:

```
CREATE SEQUENCE <nomeSq> [INCREMENT BY #]
[START WITH #]
[MINVALUE #|NOMINVALUE] [MAXVALUE #|NOMAXVALUE]
[CACHE #]
```

Comando para obter próximo valor da sequência:

```
<nomeSq>.NEXTVAL
```

Comando para definir um novo valor da sequência:

```
SELECT SETVAL('<nomeSQ>',<novoValor>);
```

A figura 4.22 mostra como criar uma nova sequência através do assistente do PgAdmin3.

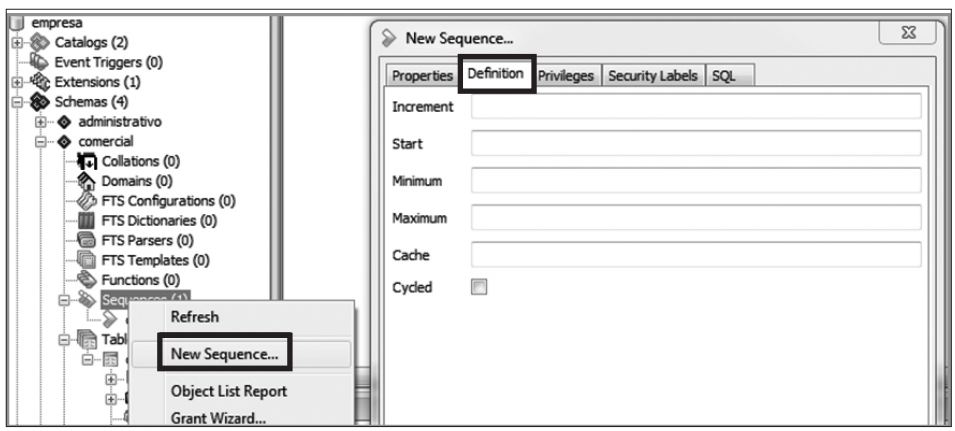


Figura 4.22 pgAdmin3 – Assistente para criação de sequências.

Exemplo de como definir uma coluna "autoincremento":

```
CREATE SEQUENCE funcionario_seq INCREMENT 1 START 1;
CREATE TABLE funcionario(
 idfunc INTEGER DEFAULT nextval('seq_func'),
...
  PRIMARY KEY(idfunc);
);
INSERT INTO funcionario VALUES (DEFAULT,'Maria');
INSERT INTO funcionario VALUES (nextval('seq_func'),'Ana');
```

Ou

```
CREATE TABLE funcionario(
 idfunc SERIAL,
...
 PRIMARY KEY(idfunc);
);
...
```

SERIAL:

- É uma máscara de dado do tipo inteiro, já com o DEFAULT definido como NEXTVAL de uma sequência cujo valor inicial é 1;
- Ao utilizar o tipo SERIAL, uma sequência de nome <tabela_campo_seq> é criada implicitamente como objeto do banco de dados;
- Os tipos podem ser:

smallserial	2 bytes	small autoincrementing interger	1 to 32767
serial	4 bytes	autoincrementing interger	1 to 214783647
bigserial	8 bytes	large autoincrementing interger	1 to 9223372036854775807

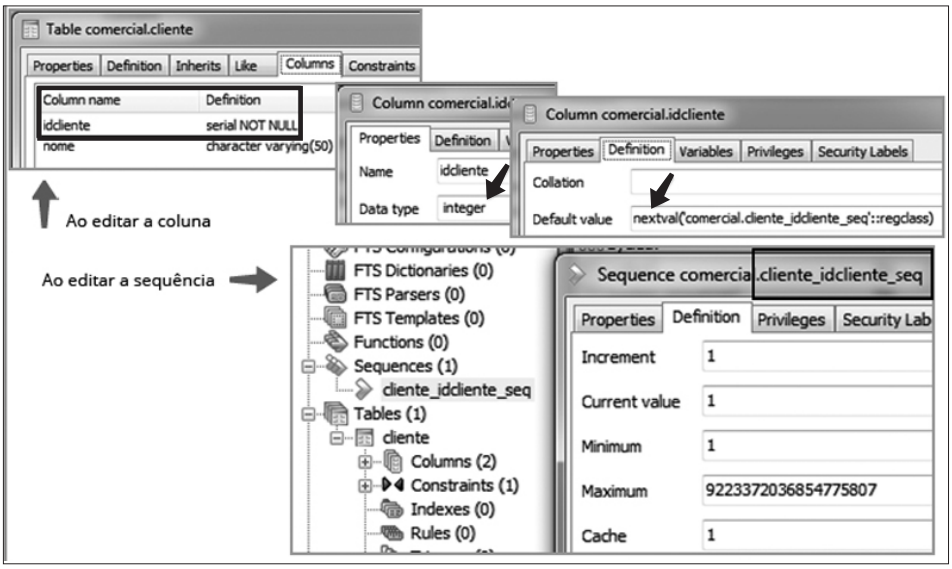


Figura 4.23
pgAdmin3 – Exemplo de atributo SERIAL na criação de uma tabela no BD.

Desvantagens no uso de seriais:

- Caso haja uma falha no sistema, será perdida toda a numeração sequencial;
- Caso você execute o comando *DELETE* na sua tabela, o número da sequência que você deletou será perdido, permanecendo um "buraco" na sua coluna;
- Não é aproveitado para um novo registro.

Optar por autonumeração quando não houver atributo identificador próprio (nesse caso, usar a cláusula UNIQUE).

Alteração da sequência:

```
ALTER SEQUENCE <nomeSQ> <parâmetros>;
```

Exclusão de uma sequência:

```
DROP SEQUENCE <nomeSQ> [CASCADE|RESTRICT];
```

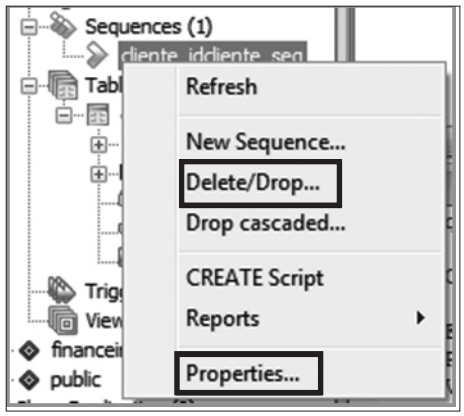


Figura 4.24
pgAdmin3 – Assistente para manutenção de sequências.

Script SQL/DDL do banco de dados

A última etapa do processo de conversão do modelo lógico em físico é a escrita do script SQL/DDL com posterior execução deste dentro do banco de dados criado dentro SGBD PostgreSQL. Em alguns casos, se forem feitos em ferramentas CASE, a geração do script SQL/DDL é possível por meio de assistentes de geração de código SQL/DDL, como no brModelo. A figura 4.25 mostra a janela do Editor de SQL do pgAdmin3.

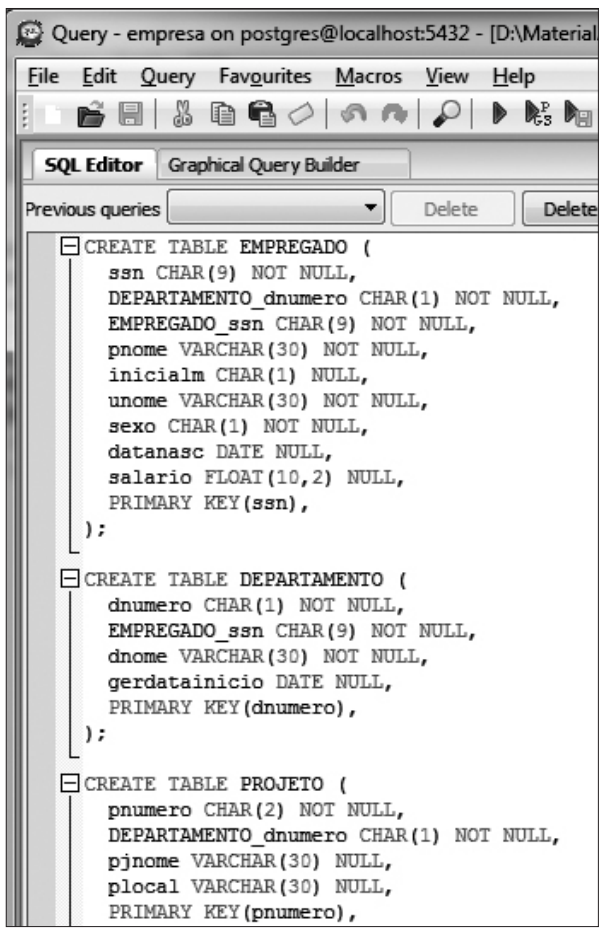


Figura 4.25
pgAdmin3 SQL Editor – Assistente para execução do scritp SQL/DDL.

Exercícios de Fixação
Comandos DDL para criação do banco de dados

Para você, qual é a importância da criação do modelo conceitual e lógico como base para a criação do modelo físico e posterior scritp SQL/DDL a ser utilizado na criação do banco de dados da aplicação em desenvolvimento?

5

Comandos DML – CRUD e operações sobre conjuntos

objetivos

Conhecer o processo de escrita de consultas SQL/DML para a manipulação de banco de dados com operações CRUD e operações sobre conjuntos.

conceitos

CRUD; INSERT; SELECT; cursor; UPDATE; DELETE; DISTINCT; JOIN (INNER e OUTER); UNION; INTERSECT e EXCEPT.

Operações CRUD

A partir desse momento, vamos dar início ao estudo dos recursos disponibilizados nos SGBDs para manipulação das estruturas de dados definidas nas tabelas que compõem um banco de dados. O conjunto de comandos está definido dentro do SQL/DML.

DML é o grupo de comandos dentro da linguagem SQL utilizado para a recuperação, inclusão, remoção e modificação de informações em bancos de dados. O acrônimo CRUD é usado frequentemente para definir as quatro operações básicas de um banco de dados. Seu significado é:

- Create (INSERT);
- Retrieve (SELECT);
- Update (UPDATE);
- Delete (DELETE).

Os comandos SQL/DML, para manipular o BD, podem ser informados através da janela do SQL Editor, sendo o resultado exibido no Output Pane, conforme pode ser visto na figura 5.1.

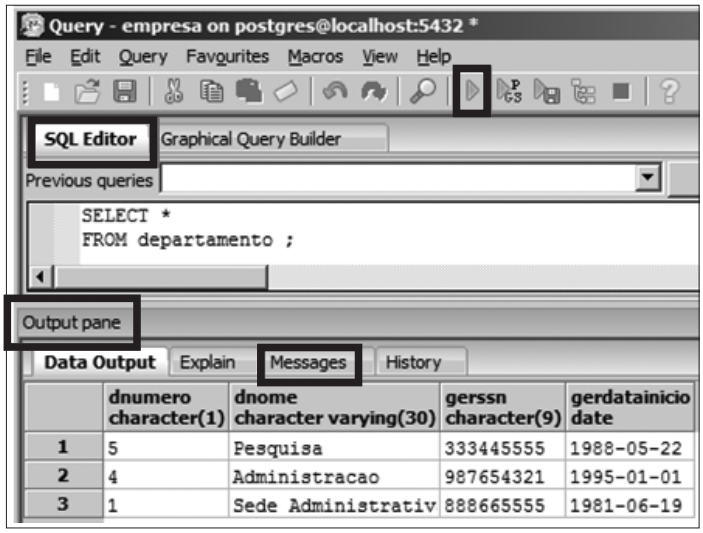


Figura 5.1
pgAdmin3 SQL Editor – resultado de uma operação CRUD no Output Pane.

Os dados obtidos através de um desses comandos, por sua vez, podem ser diretamente manipulados por meio da janela de Edição de Dados, conforme demonstrado na figura 5.2:

Edição de Dados

Exemplo tabela Empregado
(BD Empresa)

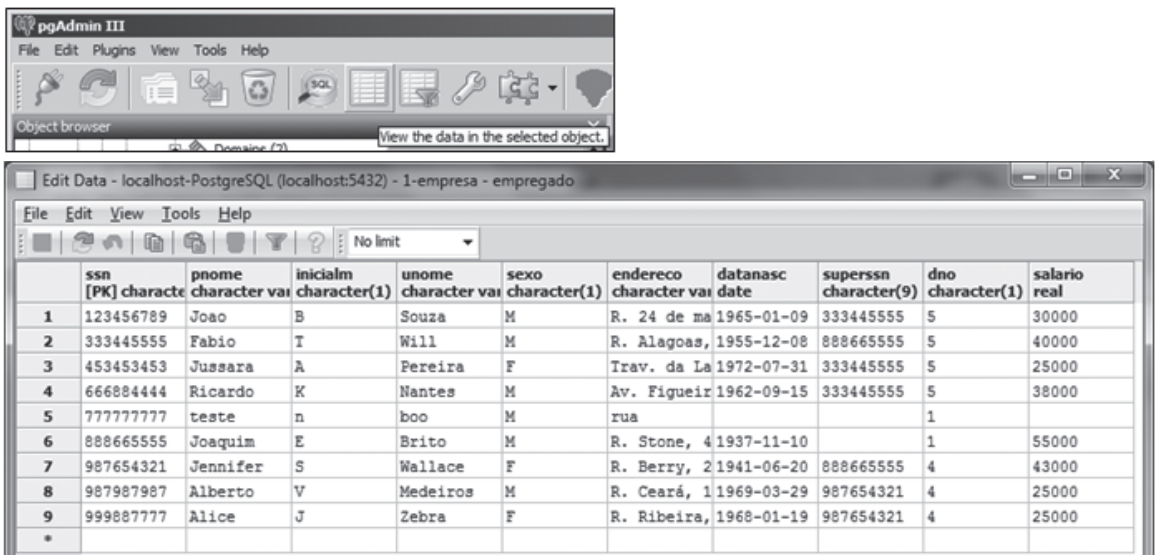


Figura 5.2
pgAdmin3 SQL Editor – editando dados de uma tabela no assistente Edit Data.

INSERT

- *INSERT*: Retorna número de registros inseridos ou código de erro

```
INSERT INTO <nomeTabela> [(<coluna1>,...)] VALUES (<expressão>,...);
```

- Observações:
- Quando o nome das colunas não é informado, utiliza-se a ordem das colunas na sua criação.
- Validação de Regras/Restrições:
  - Validação do tipo de dado (domínio): para cada tipo de dado, a expressão (valor a ser inserido) deve ser adequada.

O comando *INSERT* serve para inserir um registro em alguma tabela de um banco de dados. Todos os campos não nulos devem ser informados e os demais são de caráter facultativo.

A cláusula que identifica os nomes das colunas poderá ser suprimida. Nesse caso, o conjunto de valores da cláusula value deverá ser ordenado de acordo com a sequência de campos, quando da definição da estrutura da tabela no banco de dados.

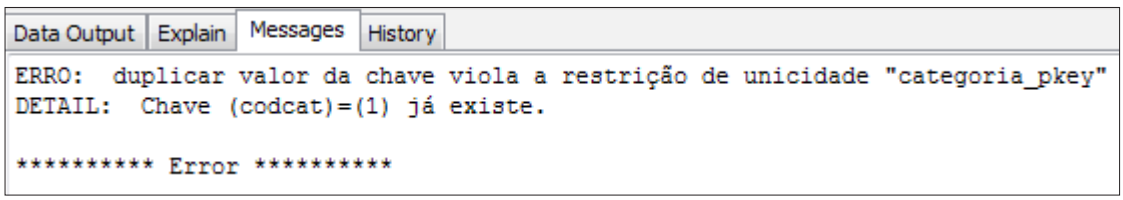
Uso do campo serial

O uso do campo serial, ou autonumerado, pode diferir dependendo do SGBD utilizado. A seguir, temos um exemplo do uso do campo serial:

```
INSERT INTO categoria (codCat, descrCat)
 VALUES (1,'informática');
INSERT INTO categoria VALUES (DEFAULT,'vestuário');
```

Figura 5.3 Mensagem de erro durante processo de inclusão de registro.

Vejam que a execução em sequência desses comandos gera um erro, conforme pode ser visto na figura 5.3. Esse erro ocorre porque já existe na tabela um registro com o valor 1 (valor da sequência). Na tentativa seguinte para inserir um registro, este vai conter o valor 2, pois a sequência já foi incrementada (executou NEXTVAL).



Nesse outro exemplo, apresentado na figura 5.4, demonstra-se que o problema pode acontecer em qualquer momento, e não apenas na inclusão dos primeiros registros em uma tabela.

Figura 5.4 Mensagem de erro – violação de restrição de unidade durante processo de inclusão de registro.

```
INSERT INTO categoria
 VALUES (NEXTVAL('categoria_codcat_seq'), 'frutas');
INSERT INTO categoria (codCat, descrCat)
 VALUES (3, 'brinquedo');
```

```
ERRO: Duplicar o valor da chave viola a restrição de unidade “categoria_pkey”
DETAIL: Chave (codcat) = 3 já existe.
```

Validação de chave primária

Ao incluir um novo registro, é preciso lembrar que devemos sempre informar a chave primária, principalmente quando fornecemos apenas os valores dos campos que serão incluídos. No exemplo a seguir, ocorre um erro quando a chave primária é omitida.

```
INSERT INTO produto VALUES
  ('abacaxi' , 4.58, 6);
```

```
ERRO: sintaxe de entrada é inválida para interger: “abacaxi”
LINE 2:   ('abacaxi', 4.58, 6)
           ^
********** Error **********
```

Podemos corrigir essa instrução incluindo a indicação do próximo valor para a PK ou informando a sequência de campos nos quais serão gravadas as informações da cláusula value. Vejamos exemplos das duas alternativas:

```
INSERT INTO produto VALUES
  (NEXTVAL('produto_codprod_seq'), 'abacaxi' , 4.58, 6);
INSERT INTO produto VALUES
  (DEFAULT, 'abacaxi', 4.58, 6);
```

Ou

```
INSERT INTO produto (descrProd, vlrProd, codCat) VALUES
('abacaxi', 4.58, 6);
```

Validação da chave estrangeira

O SGBD verifica também se existe o valor PK da tabela relacionada na tabela de origem e, caso não exista, retorna uma mensagem de erro informando que não existe o valor para a chave estrangeira naquela instrução INSERT. Vejamos um exemplo onde esse tipo de erro acontece:

```
INSERT INTO produto
  (codProd, descrProd, vlrProd, codCat) VALUES
  (DEFAULT, 'abacaxi', 4.58, 30);
```

```
ERRO: inserção ou atualização em tabela “produto” viola restrição de chave
estrangeira “produto_codcast_fkey”
DETAIL: Chave (codcat) = 30 não está presentte na tabela ”categoria”
```

Validação da restrição NOT NULL

Outra validação feita pelo PostgreSQL em um comando INSERT é sobre a possibilidade de um ou mais campos poderem ser suprimidos ou informados com valor nulo. No exemplo a seguir, temos uma situação onde um erro é gerado por conta dessa validação.

```
INSERT INTO produto (codProd, vlrProd, codCat) VALUES (DEFAULT, 2.5, 3);
```

```
ERRO: valor nulo na coluna “descripod” viola a restrição não-nula
DETAIL: Failing row contains (1, null, null, null, null, null, null, 3, 2.5.
********** Error **********
```

Nesse exemplo, é obrigatório informar o campo “descrprod” ou passar um valor para ele na ordem de definição dos campos quando da criação da tabela no banco de dados.

```
INSERT INTO produto
 (codProd, descrProd, vlrProd, codCat) VALUES
 (DEFAULT, 'abacaxi', 4.58, 30);
```

Validação da restrição de domínio (tipo de dado)

Como é possível determinar um domínio para campos em uma tabela, o postgreSQL consulta a sua estrutura para verificar se existem restrições de domínio durante a execução do comando *INSERT*. No exemplo a seguir, foi informada a restrição para o campo “sexo”, onde este só vai aceitar M ou F, gerando um erro já que tal domínio não foi observado.

```
INSERT INTO empregado
(ssn, pnome, inicialm, unome, sexo, dno)
VALUES ('123454321', 'Denis', 'C', 'Neves', 'P', '1');
```

```
ERRO: novo registro da relação “empregado” viola restrição de verificação
“empregado_sexo_check”
DETAIL: Failing row contains (123454321, Denis, C, neves, P, null, null, null, 1, null,
null, null).
********** Error **********
```

Outras validações são feitas em relação a campos do tipo “DATE” e “BOOLEAN”. Vejamos alguns dos formatos que são aceitos pelo PostgreSQL:Q

- Tipo DATE pode ser no formato “AAAA-MM-DD” ou “DD/MM/AAAA”, ou ainda a data do sistema: current_date;
- Tipo BOOLEAN pode ser nos formatos:

TRUE	FALSE
‘t’	‘f’
‘true’	‘false’
‘y’	‘n’
‘yes’	‘no’
‘on’	‘off’
‘1’	‘0’

UPDATE

O comando *UPDATE* serve para realizar a alteração de algum registro já inserido em alguma tabela de um banco de dados. Esse comando retorna número de registros afetados pela atualização na tabela ou, então, um código de erro. A sintaxe do comando é:

```
UPDATE <nomeTabela>
  SET <campo1> = <expressão1> [,...]
  [WHERE <expressãoCondicional>];
```

Vejamos alguns exemplos:

```
UPDATE produto SET precoProd = precoProd*1.15;
UPDATE funcionario SET idade = 35, salario = 1890
 WHERE codfunc = 3;
UPDATE produto SET codCat = 10
 WHERE codProd = 3;
```

```
ERRO: inserção ou atualização em tabela "produto" viola restrição de chave
estrangeira "produto_codcat_fkey"
DETAIL: Chave (codcat)=(10) não está presente na tabela "categoria".
```

Chamamos a atenção para problemas de violação de chave estrangeira, devendo ser informado um valor existente de PK da tabela relacionada também no processo de alteração dos valores de um registro em uma tabela do banco de dados.

DELETE

O comando *DELETE* apaga um determinado registro ou um conjunto de registros que satisfaçam uma determinada condição. Esse comando retorna o número de registros removidos da tabela ou um código de erro, sendo a sua sintaxe apresentada a seguir.

```
DELETE FROM <nomeTabela>
  [WHERE <expressãoCondicional>];
```

Exemplos:

```
DELETE FROM aluno;
DELETE FROM funcionario WHERE codfunc = 3;
DELETE FROM categoria WHERE codcat = 1;
```

```
ERRO: atuailzação ou exclusão em tabela "categoria" viola restrição de chave
estrangeira "produto_coscat_fkey" em "produto"
DETAIL: ahve (codcat) = (1) ainda é referenciada pela tabela "produto"

********** Error **********
```

Saiba mais

O problema de violação de chave estrangeira também pode ocorrer na execução de um comando ***DELETE***, conforme pudemos ver no exemplo.

TRUNCATE

Esse comando exclui todas as linhas de uma tabela ou de uma lista de tabelas de forma mais rápida que o comando *DELETE*, apresentando ganho de performance por não permitir a inclusão de restrições com o uso da cláusula WHERE. O *TRUNCATE* remove rapidamente todas as linhas da tabela e recupera espaço em disco imediatamente (muito útil em tabelas grandes). Sua sintaxe é:

```
TRUNCATE [TABLE] [ONLY] <nomeTB> [, ... ]
[RESTART IDENTITY|CONTINUE IDENTITY]
[CASCADE|RESTRICT];
```

SELECT

É o comando DML utilizado para manipular os dados armazenados nas estruturas das tabelas de um banco de dados, retornando os campos desejados de todos os registros que satisfazem a condição estabelecida. O resultado é apresentado na forma de um cursor, que é uma tabela temporária que pode ser entendida como uma matriz em que cada linha contém um registro e cada coluna corresponde a um campo indicado no comando *SELECT*, que deu origem ao cursor. A sintaxe desse comando é apresentada a seguir.

```
SELECT <listaCampos>
   FROM <listaTabelas>
   [WHERE <expressãoCondicional>]
   [...];
```

Na figura 5.5, temos um exemplo simples do comando *SELECT* juntamente com o cursor que é gerado através da sua execução.

```
SELECT pnome, endereco FROM empregado;
```

Data Output | Explain | Messages | History

	pnome character v.	endereco character varying(50)
1	Joao	R. 24 de maio, 1500 - Curiti
2	Fabio	R. Alagoas, 325 - Curitiba -
3	Alice	R. Ribeira, 98 - Pinhais - P
4	Jennifer	R. Berry, 291 - Colombo - PR
5	Ricardo	Av. Figueira, 55 - Almirante

Figura 5.5
Tabela temporária com o resultado de uma consulta SELECT.

Outros Exemplos:

```
SELECT descrProd FROM produto
  WHERE vlrProd >= 2000;

SELECT * FROM produto
  WHERE codCat = 2 AND vlrProd < 500;
SELECT * FROM trabalha WHERE horas > 20;
```

Operadores

Resumo dos principais operadores:

- Asterisco (*): lista todos os campos.
  - Exemplo: **SELECT** * **FROM** produto**;**
- Aritméticos: +, -, *, /
- Aplicados no SELECT ou no WHERE sobre campos numéricos.
- Relacionais: =, <>, >, >=, <, <=
- Lógicos: AND, OR, NOT.

Existem funções especializadas na especificação SQL (manipulação de string, data/hora, números/estatísticas), implementadas de formas diferentes nos SGBDs.

Operadores são usados dentro da estrutura do comando *SELECT* para possibilitar, entre outras, a filtragem de dados ou condições de restrições. Geralmente são utilizados para possibilitar a busca por dados que satisfaçam determinadas situações onde não há a necessidade de visualização de todos os dados gravados nas tabelas de um determinado banco de dados.

Um operador interessante é o LIKE, que permite a comparação entre substrings. Esse operador trabalha com o %, que funciona como uma espécie de coringa na definição de partes de um substring. Os exemplos a seguir ilustram melhor o funcionamento desse operador:

- WHERE nome **LIKE** '%ANA%' (contém ANA)
- WHERE nome **LIKE** 'AN%' (começam com AN)
- WHERE nome **LIKE** '%M' (terminam com M)
- WHERE nome **LIKE** '_A%' (contém a 2ª letra A)
- WHERE nome **LIKE** '_ _ _' (contém 3 caracteres)
- contém _ ou %: WHERE nome **LIKE** '%_%'
- SELECT * FROM empregado
  - **WHERE** nome **LIKE '%ana%'**;

Nota: no PostgreSQL, há diferença de maiúscula e minúscula na pesquisa com string.

Outros operadores que podem ser úteis em ocasiões específicas são o "BETWEEN...AND...", "IS NULL" e "IN". A seguir, descrevemos rapidamente cada um deles com seus respectivos exemplos de utilização.

- Operador "BETWEEN...AND...": comparações com valor numérico
  WHERE salario BETWEEN 3000 AND 4000;
- Operador "IS NULL": é um valor nulo
  SELECT nome FROM empresa
  WHERE contato IS NULL;
- Operador "IN": comparação com uma lista de valores
  SELECT * FROM empregado
  WHERE salario IN (500, 700, 200);
  ... WHERE nome IN ('JOANA', 'ANA', 'MARIA');

Operadores negativos

É importante destacar o uso dos operadores de negação, que podem ser aplicados juntamente com os operadores anteriormente descritos, estabelecendo restrições que trazem como resultado tudo o que NÃO satisfaz a condição estabelecida. A seguir apresentamos exemplos de como usar esse tipo de operador.

- <>: diferente

```
WHERE salario <> 1243;
```

- NOT <nomeCampo> =: diferente do campo

```
WHERE salario NOT = 1243;
```

- NOT <nomeCampo> >: não maior que

```
WHERE salario NOT > 1243;
```

- NOT BETWEEN...AND...: não entre dois valores informados

```
WHERE salario NOT BETWEEN 3000 AND 4000;
```

- NOT IN (<valor1>,...): não existente em uma dada lista de valores

```
WHERE nome NOT IN ('JOANA', 'ANA', 'MARIA');
```

- NOT LIKE "...": diferente do padrão de caracteres informado

```
WHERE nome NOT LINE '%ana%';
```

- IS NOT NULL: não é um valor nulo

```
WHERE contato IS NOT NULL;
```

Comandos especiais

Apesar de inicialmente parecer ser um comando simples, o *SELECT* admite uma série de possibilidades para permitir que seu uso seja bastante amplo. Assim, junto com o *SELECT*, podem ser aplicados comandos especiais, conforme podemos ver nos exemplos a seguir.

- **Cálculo** no SELECT

Exemplo: Exibir salário com aumento de 10%

```
SELECT salario*1.1 FROM funcionario;
```

- Uso de **apelidos** em campo com **AS**

```
SELECT salario AS Atual, salario*1.1 AS "Novo Salário"
FROM empregado;
```

- Remoção de valores duplicados em uma coluna com DISTINCT

```
SELECT DISTINCT cidade FROM cliente;
```

ORDER BY

Outro aspecto a ser considerado é como ordenar o resultado de um comando *SELECT* qualquer. Para isso, temos a cláusula ORDER BY, que admite algumas variações para indicar a ordenação que deve ser seguida e que apresenta as seguintes características:

- Ordena por um ou mais campos especificados (usar como critério de desempate);
- De modo crescente (ASC-padrão) ou decrescente (DESC);
- Pode ser especificado um número representando a ordem de definição no SELECT.

A figura 5.6 traz um exemplo do uso da clausula ORDER BY e o cursor gerado refletindo a ordenação determinada.

```
SELECT pnome, salario
FROM empregado
ORDER BY pnome; -- ORDER BY 1
```

pnome	salario
Alberto	25000
Alice	25000
Fabio	40000
Jennifer	43000
Joao	30000
Joaquim	55000
Jussara	25000

Figura 5.6
Tabela temporária com o resultado de uma consulta SELECT ordenada por pnome.

É importante destacar que é possível ordenar o resultado de acordo com o resultado de funções aplicadas nos dados selecionados, e que provavelmente não ficarão imediatamente aparentes. Um exemplo disso é apresentado na figura 5.7.

Exemplo: ordenar pelos quatro **últimos** dígitos do SSN

- Uso da função nativa RIGHT(expressão, nº char)

```
SELECT pnome, ssn
   FROM empregado
   ORDER BY RIGHT(ssn,4);
```

pnome	ssn
Jennifer	987654321
Ricardo	666884444
Jussara	453453453
Fabio	333445555
Joaquim	888665555
Alice	999887777
Joao	123456789
Alberto	987987987

pnome	ssn
Jussara	453453453
Jennifer	987654321
Ricardo	666884444
Fabio	333445555
Joaquim	888665555
Joao	123456789
Alice	999887777
Alberto	987987987

Figura 5.7
Resultado de um SELECT ordenado por parte do ssn.

LIMIT

A cláusula LIMIT, como o próprio nome indica, impõe um limite para a quantidade de registros retornados por um comando *SELECT*. Assim, o cursor gerado conterá no máximo o número de registros indicados pela cláusula, sendo por isso mesmo comumente utilizada em "consultas de ranqueamento". Esta pode ser utilizada em conjunto com a cláusula OFFSET <início>, que permite desconsiderar uma quantidade determinada dos registros inicialmente retornados pela consulta. Na figura 5.8, temos o exemplo de um SELECT que vai consultar os três maiores salários da tabela empregado, juntamente com o resultado obtido.

Figura 5.8
Tabela temporária com o resultado de uma consulta SELECT com LIMIT.

```
SELECT pnome, salario
FROM empregado
ORDER BY salario DESC
```

Data Output | Explain | Messages | History

	pnome character varying(30)	salario double precision
1	Joaquim	55000
2	Jennifer	43000
3	Fabio	40000

Atividade de fixação 

Junção de tabelas

A operação de junção usa a combinação de tuplas, que indicam um relacionamento entre tabelas em um banco de dados, para possibilitar a execução de consultas SQL que necessitam produzir resultados com dados de oriundos de duas ou mais tabelas.

A ligação ou vínculo entre duas ou mais tabelas é feita através da chave primária de uma tabela e das chaves estrangeiras das demais. É preciso, contudo, especificar a condição de junção e o uso de apelidos para as tabelas envolvidas pode ajudar a deixar o comando *SELECT* mais fácil de ser entendido. Na figura 5.9, temos um exemplo onde se deseja listar o departamento e os empregados que nele estão lotados, ordenando o resultado por departamento.

```
SELECT d.dnome, e.pnome
FROM empregado e, departamento d
WHERE e.dno = d.dnumero
ORDER BY d.dnome, e.pnome;
```

	dnome character varying(30)	pnome character v
1	Administracao	Alberto
2	Administracao	Alice
3	Administracao	Jennifer
4	Pesquisa	Fabio
5	Pesquisa	Joao
6	Pesquisa	Jussara
7	Pesquisa	Ricardo
8	Sede Administrativ	Catia
9	Sede Administrativ	Denis
10	Sede Administrativ	Fabio
11	Sede Administrativ	Joaquim
12	Sede Administrativ	Sergio

A partir da publicação do padrão SQL-92, a sintaxe da linguagem passou a admitir o uso da cláusula JOIN, trazendo mais portabilidade e legibilidade para as operações de junção de múltiplas tabelas, já que dessa forma as condições da junção ficam separadas das condições de filtro (WHERE).

Tipos de Junção (JOIN):

- **Interna** (INNER): serão incluídas somente as linhas que satisfazem a condição de junção. Também pode ser Cruzada (CROSS) ou Natural (NATURAL, servidor determina a junção);
- **Externa** (OUTER): serão incluídas linhas que satisfaçam a condição de junção e as **linhas restantes** de uma das tabelas da junção. Pode ser RIGHT, LEFT ou FULL.

A seguir, são analisados os diferentes tipos de JOIN existentes.

Junção Interna

A Junção Interna (INNER JOIN) é utilizada para especificar forma de junção com uso de chave estrangeira e da subcláusula ON. Se não há dado de relação, então a tupla não aparece no resultado (se necessário mostrar, usar junção externa). A sintaxe do SELECT com esse tipo de junção é:

```
SELECT <listaCampos>
FROM <Tabela1>
INNER JOIN <Tabela2> ON t1.<pk> = t2.<fk>
[INNER JOIN <Tabela3> ON t3.<pk> = <t2.fk>...];
```

Na figura 5.10, são apresentados alguns exemplos do uso da cláusula INNER JOIN e o comando *SELECT* equivalente utilizando apenas a cláusula WHERE.

Junção de Tabelas

Exemplos:

```
SELECT c.descrCat, p.descrProd
FROM categoria c
INNER JOIN produto p ON c.codCat = p.codCat ;
no lugar de
SELECT c.descrCat, p.descrProd
FROM categoria c, produto p
WHERE c.codCat = p.codCat;
```

Categoria (@**codCat**, descricao)
Produto(@**codProd,** descricao, @**codCat)**
Cliente(@**codCli**, nome)
pedido(@**codPedido**, valorTotal, @**codCli**)

```
SELECT DISTINCT a.nomealu
FROM matricula m
INNER JOIN aluno a ON m.codalu = a.codalu
WHERE m.faltas = 0;
no lugar de
SELECT DISTINCT a.nomealu
FROM matricula m, aluno a
WHERE m.faltas = 0 AND m.codalu = a.codalu;
```

Professor (@**codProf**, nomeProf)
Aluno(@**numAlu**, nomeAlu)
Disciplina(@**codDisc**, descDisc, @**codProf**)
matricula(@**numAlu**, @**codDisc**, nota, faltas)

Figura 5.9
Exemplo de INNER JOIN.

Se os nomes dos campos forem iguais nas tabelas, pode-se usar a subcláusula USING, conforme o seguinte exemplo:

```
SELECT c.descrCat, p.descrProd
FROM categoria c
INNER JOIN produto p USING (codCat);
```

Outro ponto a ser lembrado é que a junção interna será sempre feita por padrão, sendo portanto opcional utilizar INNER no comando.

```
SELECT c.descrCat, p.descrProd
FROM categoria c
[INNER] JOIN produto p USING (codCat);
```

Junção Cruzada

A Junção Cruzada, por sua vez, corresponde ao produto cartesiano das tabelas envolvidas. Ela é a junção realizada quando não se especifica ON no comando. Não é um tipo de junção utilizado com frequência, mas muitas vezes é realizada justamente pelo esquecimento do uso do ON no comando. A figura 5.11 traz um exemplo de junção cruzada.

```
SELECT c.descrCat, p.descProd
FROM categoria c
[CROSS] JOIN produto p;
```

descrCat	descProd
LIMPEZA	BOMBRIL
VESTUARIO	BOMBRIL
LATICINIO	BOMBRIL
ACOUGUE	BOMBRIL
FARINACEO	BOMBRIL
INFORMATICA	BOMBRIL
LIVRARIA	BOMBRIL
PAPELARIA	BOMBRIL
LIMPEZA	DESINFETANTE
VESTUARIO	DESINFETANTE
LATICINIO	DESINFETANTE
ACOUGUE	DESINFETANTE
FARINACEO	DESINFETANTE
INFORMATICA	DESINFETANTE
LIVRARIA	DESINFETANTE
PAPELARIA	DESINFETANTE

Figura 5.10
Tabela temporária com o resultado de uma junção cruzada entre duas tabelas.

Junção natural

A Junção Natural tem esse nome por deixar para o SGBD determinar as condições de junção. A figura 5.12 traz um exemplo de junção natural.

```
SELECT c.descrcat, p.descprod
FROM categoria c
NATURAL JOIN produto p;
```

descrcat	descprod
LIMPEZA	BOMBRIL
LIMPEZA	DESINFETANTE
LIMPEZA	SABÃO EM PÓ
VESTUARIO	CALÇA
VESTUARIO	CAMISETA
VESTUARIO	VESTIDO
LATICINIO	YAKULT
LATICINIO	IOGURTE DANON
ACOUGUE	COXÃO MOLE
ACOUGUE	MOÍDA DE 1A
ACOUGUE	FILÉ MIGNON
FARINACEO	CEREAL MATINA
INFORMATICA	MONITOR LCD
INFORMATICA	HD EXTERNO
INFORMATICA	PEN DRIVE 8 GB

Figura 5.11
Tabela temporária com o resultado de uma junção natural entre duas tabelas.

Autojunções

Esse é um recurso utilizado quanto temos um autorrelacionamento no modelo físico do banco de dados. Ou seja, quando é preciso fazer uma junção com dados de uma mesma tabela (relacionamento unário). A figura 5.13 traz um exemplo desse tipo.

```
SELECT s.pnome AS supervisor,
    e.pnome AS supervisionado
FROM empregado e
INNER JOIN empregado s ON s.ssn = e.superssn
ORDER BY s.ssn;
```

supervisor	supervisionado
Fabio	Ricardo
Fabio	Jussara
Fabio	Joao
Joaquim	Fabio
Joaquim	Jennifer
Jennifer	Alice
Jennifer	Alberto

Figura 5.12
Tabela temporária com o resultado de uma autojunção.

Junção de três ou mais tabelas

A junção de três ou mais tabelas é feita através do uso sucessivo das cláusulas INNER JOIN <tabela> ON <condiçãoJunção>. A ordem das tabelas no JOIN é irrelevante, já que a escolha da "tabela condutora" é feita pelo SGBD. Na figura 5.14, temos um exemplo da junção de três tabelas.

```
SELECT d.dnome, p.pjnome, t.essn, e.pnome
FROM projeto p
INNER JOIN departamento d ON p.dnum = d.dnumero
INNER JOIN trabalha t ON p.pnumero = t.pno
INNER JOIN empregado e ON e.ssn = t.essn;
```

	dnome character varying(30)	pjnome character varying(30)	essn character(9)	pnome character v
1	Pesquisa	ProdutoX	123456789	Joao
2	Pesquisa	ProdutoY	123456789	Joao
3	Pesquisa	ProdutoZ	666884444	Ricardo
4	Pesquisa	ProdutoX	453453453	Jussara
5	Pesquisa	ProdutoY	453453453	Jussara
6	Pesquisa	ProdutoY	333445555	Fabio
7	Pesquisa	ProdutoZ	333445555	Fabio
8	Administracao	Automatizacao	333445555	Fabio
9	Sede Administrativ	Reorganizacao	333445555	Fabio
10	Administracao	Novos Beneficios	999887777	Alice
11	Administracao	Automatizacao	999887777	Alice
12	Administracao	Automatizacao	987987987	Alberto
13	Administracao	Novos Beneficios	987987987	Alberto
14	Administracao	Novos Beneficios	987654321	Jennifer

Figura 5.13
Tabela temporária com o resultado de uma junção entre três tabelas.

Caso você esteja se perguntando, é possível, sim, informar a mesma tabela mais de uma vez em uma mesma consulta. O exemplo a seguir demonstra o uso de uma mesma tabela duas vezes.

```
SELECT <campos>
FROM <tabela_ta> tA
INNER JOIN <tabelaT> t1  ON tA.chave = t1.campoT
INNER JOIN <tabela_tb> tB ON tA.chave = tB.chave
INNER JOIN <tabelaT> t2  ON tB.chave = t2.campoT;
```

Outra possibilidade é o uso de subconsultas na construção de uma tabela temporária, conforme demonstrado no seguinte exemplo:

```
SELECT <campos>
FROM <tabela t>
INNER JOIN (subconsulta1) s1 ON t.chave = s1.campo1S1
INNER JOIN (subconsulta2) s2 ON s1.campo2S2 = s2.campo;
```

Veja que subconsulta1 e subconsulta2 representam instruções SQL que irão retornar seus respectivos cursores (tabelas temporárias) com informações necessárias para retornar a tabela temporária final desejada.

Junções equivalentes x não equivalentes

Junções não equivalentes são aquelas que não fazem uso de chave estrangeira. A seguir são apresentados exemplos para ajudar a entender esse conceito.

Exemplo 1: encontrar funcionários que começaram a trabalhar na loja enquanto um produto 'X' estava sendo oferecido:

```
SELECT e.nome, e.dtinicio
FROM empregado e
INNER JOIN produto p
 ON e.dtini >= p.dtini AND e.dtini <= p.dtfim
WHERE pnome = 'X';
```

Exemplo 2: formar um torneio de xadrez de modo que um empregado não jogue com ele mesmo.

```
SELECT e1.pnome AS competidor1,'VS',
    e2.pnome AS competidor2
FROM empregado e1
INNER JOIN empregado e2
  ON e1.ssn > e2.ssn;
```

A figura 5.15 apresenta o curso resultante da execução do *SELECT* do exemplo 2 anterior. Vejam que são oito empregados, mas uma mesma pessoa não deve jogar consigo mesmo. Assim, teríamos 8x7 = 56 registros. Mas uma partida entre comp1 x comp2 é equivalente a uma partida entre comp2 x comp1. Desse modo, o resultado final deve ter 28 registros (ou partidas). Para alcançar esse resultado, é necessário especificar que e1.ssn > e2.ssn.

competidor1	VS	competidor2
Alberto	VS	Joao
Alberto	VS	Ricardo
Alberto	VS	Jennifer
Alberto	VS	Alice
Alberto	VS	Joaquim
Alberto	VS	Fabio
Alberto	VS	Jussara
Alice	VS	Joao
Alice	VS	Jussara
Alice	VS	Ricardo
Alice	VS	Jennifer
Alice	VS	Joaquim
Alice	VS	Fabio
Alice	VS	Alberto
Fabio	VS	Ricardo
Fabio	VS	Jennifer
Fabio	VS	Joaquim
Fabio	VS	Alice
Fabio	VS	Alberto
Fabio	VS	Joao
Fabio	VS	Jussara
Jennifer	VS	Fabio

competidor1	VS	competidor2
Alberto	VS	Fabio
Alberto	VS	Joaquim
Alberto	VS	Joao
Alberto	VS	Ricardo
Alberto	VS	Jennifer
Alberto	VS	Jussara
Alice	VS	Fabio
Alice	VS	Ricardo
Alice	VS	Jussara
Alice	VS	Alberto
Alice	VS	Joao
Alice	VS	Jennifer
Alice	VS	Joaquim
Fabio	VS	Joao
Jennifer	VS	Fabio
Jennifer	VS	Ricardo
Jennifer	VS	Jussara
Jennifer	VS	Joao
Jennifer	VS	Joaquim
Joaquim	VS	Jussara
Joaquim	VS	Fabio
Joaquim	VS	Joao
Joaquim	VS	Ricardo
Jussara	VS	Joao
Jussara	VS	Fabio
Ricardo	VS	Joao
Ricardo	VS	Fabio
Ricardo	VS	Jussara

Figura 5.14
Tabela temporária com o resultado de uma junção não equivalente.

Junção externa

Tipos de OUTER JOIN:

- A palavra-chave LEFT especifica que a tabela do lado esquerdo é responsável por determinar o número de linhas do conjunto-resultado.
- A palavra-chave RIGHT especifica que a tabela do lado direito é usada para fornecer valores de consulta sempre que uma correspondência é encontrada.
- Nas pesquisas com FULL OUTER JOIN o resultado trará todos os registros, ao menos uma vez, que estejam nas dois tabelas, tanto a da esquerda do JOIN quanto a da direita do JOIN.

Uma junção externa (OUTER JOIN) é utilizada sempre que se deseja ver quais linhas de uma tabela estão relacionadas a outra tabela e quais linhas não estão. Por exemplo, em uma situação em que temos CLIENTES e seus PEDIDOS armazenados em um BD, pode ser necessário descobrir quais clientes têm pedido e quais não tem pedido algum.

A junção externa pode também ser útil quando se deseja verificar se existem membros órfãos em um SGBD, ou seja, tabelas cujas PK e FK estejam sem sincronia (lembrando que SGBDs mais modernos não permitem que isso venha a acontecer).

Um OUTER JOIN só pode ser realizado entre duas tabelas, mas admite algumas variações, conforme veremos a seguir.

São apresentados a seguir exemplos do uso de junções externas. O primeiro exemplo trará como resultado os clientes que tem e os que não têm pedido. Os clientes sem pedido serão aqueles onde aparece "null" ao lado dos respectivos nomes.

```
SELECT c.nome, p.codcli, p.numped
FROM cliente c
LEFT OUTER JOIN pedido c ON c.codigo = p.codcli;
```

Já o segundo exemplo, a seguir, retornará a capacidade de carga de todos os veículos na tabela. Notem que deverá aparecer "null" ao lado das placas que não são de caminhão. Esse exemplo demonstra um uso comum para o OUTER JOIN quando temos casos de generalização ou especialização.

```
SELECT v.placa, c.carga
FROM veiculo v
LEFT OUTER JOIN caminhao c ON v.placa = c.placa;
```

Operações sobre conjuntos

Operações sobre conjunto trabalham com a teoria de conjuntos implementada na álgebra relacional.

As operações da álgebra relacional são normalmente divididas em dois grupos. O primeiro deles inclui um conjunto de operações da teoria de conjuntos. As operações são UNION, INTERSECTION, DIFFERENCE e CARTESIAN PRODUCT. O segundo grupo consiste em operações desenvolvidas especificamente para bases de dados relacionais, tais como: SELECT, PROJECT e JOIN.

As Junções e Operações sobre Conjuntos ajudam a entender as operações de junção sobre tabelas vistas em 5.7. Ainda que você possa interagir com os dados em um BD uma linha por vez, a verdadeira razão de ser dos BDs relacionais está na possibilidade de enxergá-los como conjuntos, conforme exemplificado na figura 5.16.

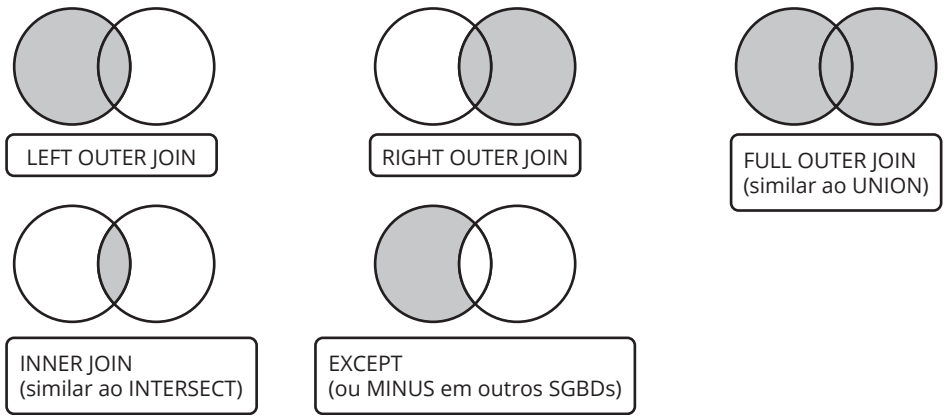


Figura 5.15
Junções sobre Conjuntos.

Em PostgreSQL, podemos combinar os resultados de duas consultas utilizando as operações de conjunto união, interseção e diferença, como apresentado na figura 5.17.

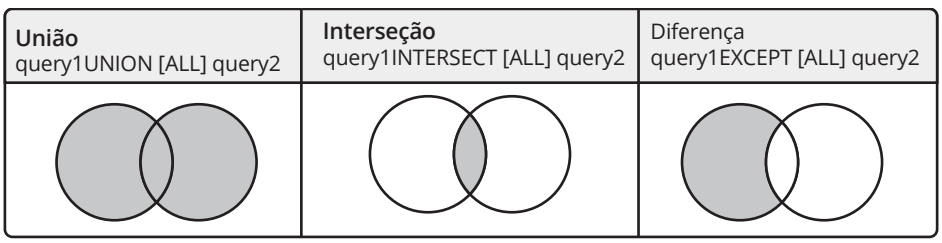


Figura 5.16
Teoria de Conjuntos.

Onde query1 e query2 são consultas que podem utilizar qualquer uma das funcionalidades estudadas até agora. As operações de conjuntos também podem ser aninhadas ou encadeadas.

No entanto, para ser possível calcular a união, a interseção, ou a diferença entre duas consultas, essas duas consultas devem ser compatíveis, ou seja, ambas devem retornar o mesmo número de colunas. Além disso, as colunas correspondentes devem possuir o mesmo tipo de dado. Essas consultas são conhecidas como compostas ou combinadas.

Consulta Composta (ou Combinada):

- Os conjuntos são representados por SELECT;
- Ambos os conjuntos devem ter o mesmo número de colunas;
- Tipos de dados de cada coluna ao longo dos dois conjuntos devem ser os mesmos (ou o servidor deve ser capaz de converter);
- Pode-se empregar ORDER BY: campos da primeira consulta;
- Precedência: INTERSECT, parênteses, de cima para baixo.

UNION

Essa operação une o resultado e remove eventuais duplicatas, com efeito semelhante ao da cláusula DISTINCT. Mas se for aplicado o comando *UNION ALL*, as duplicatas não são removidas (sendo mais "simples" para o servidor). Na figura 7.17 vemos exemplos de uso do comando *UNION*.

Exemplo 1: listar todos os locais onde existe um departamento ou um projeto.

```
SELECT plocal FROM projeto
UNION
SELECT dlocalizacao FROM localizacao;
```

plocal character varying(30)
Pinhais
Colombo
Araucaria
Curitiba

Exemplo 2: mostrar os empregados e os dependentes.

```
SELECT pnome FROM empregado
UNION ALL
SELECT nomedep FROM dependente;
```

pnome character varying(30)
Joao
Fabio
Alice
Jennifer
Ricardo
Jussara
Alberto
Joaquim
Alice
Teodoro
Joana
Abdala
Michel
Alice
Elizabete

Figura 5.17
Exemplos de UNION.

INTERSECT

Retorna os registros presentes tanto na consulta 1 quanto na 2. Se não houver dados em comum, o conjunto-resultado é vazio. As linhas duplicadas são eliminadas, a menos que INTERSECT ALL seja usado. Exemplos do uso de INTERSECT são apresentados na figura 7.19.

Exemplo 1: Listar todos os locais onde existe ambos um departamento e um projeto.

```
SELECT plocal FROM projeto
INTERSECT
SELECT dlocalizacao FROM localizacao;
```

plocal character varying(30)
Araucaria
Curitiba
Pinhais

Exemplo2: Mostrar apenas os clientes que sejam também fornecedores.

```
SELECT nome FROM cliente
INTERSECT
SELECT nome FROM fornecedor;
```

Figura 5.18
Exemplos de INTERSECT.

EXCEPT

A operação EXCEPT retorna todos os registros presentes no resultado da consulta 1, mas não no resultado da 2 (chamado às vezes de diferença entre duas consultas). Mais uma vez, as duplicatas são eliminadas, a menos que seja utilizado a cláusula EXCEPT ALL. A figura 5.20 traz exemplos utilizando essa operação.

Exemplo 1: construa uma lista de todos os locais onde existe um departamento, mas nenhum projeto.

```
SELECT dlocalizacao FROM localizacao
EXCEPT
SELECT plocal FROM projeto;
```

dlocalizacao character varying(30)
Colombo

Exemplo 2: Mostrar apenas os clientes que não sejam fornecedores

```
SELECT nome FROM cliente
EXCEPT
SELECT nome FROM fornecedor;
```

Figura 5.19
Exemplos de EXCEPT.

6

Funções agregadas e nativas

objetivos

Aprender sobre o processo de escrita de consultas SQL/DML com o uso de recursos que possibilitam a manipulação de dados de forma a organizá-los e formatá-los de acordo com a necessidade do projeto.

conceitos

Funções agregadas, agrupamento e funções nativas.

Exercício de nivelamento

Você já manipulou tabelas em banco de dados com funções agregadas ou nativas? Em caso de resposta afirmativa, você o fez como DBA ou como programador? Cite o banco de dados e/ou a linguagem de programação caso tenha utilizado, bem como algumas das funções utilizadas.

Funções agregadas

Alguns exemplos desse tipo de função são:

- AVG(expressão): retorna o valor médio de um conjunto.
- COUNT(*) ou COUNT(expressão): retorna a quantidade de valores de um conjunto.
- MAX(expressão): retorna o valor máximo dentro de um conjunto.
- MIN(expressão): retorna o valor mínimo dentro de um conjunto.
- SUM(expressão): retorna a soma dos valores de um conjunto.

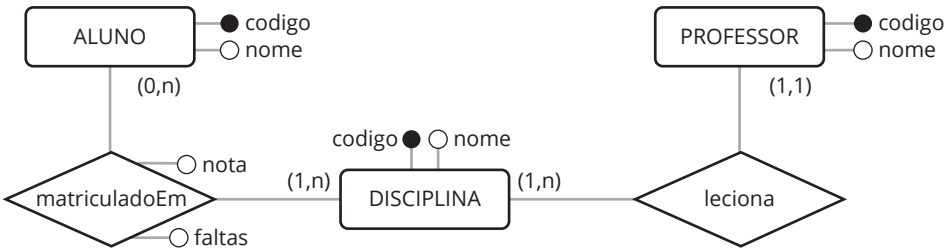
As funções agregadas ou de agrupamento não podem ser usadas na cláusula WHERE de uma consulta SQL/DML. Elas foram desenvolvidas para serem utilizadas entre a cláusula SELECT e FROM.

Funções agregadas são utilizadas para produzir resultados únicos para um conjunto de dados contendo várias linhas (tuplas) fornecidas como entrada, geralmente representando dados armazenados em tabelas. Essas funções fornecem capacidades adicionais à cláusula SELECT por realizarem cálculos com base em critérios estabelecidos.

Nos exemplos recém-citados, expressão indica qual campo será utilizado para o agrupamento que se deseja como resultado dessa consulta. As instruções ALL e DISTINCT poderão ser utilizadas para indicar se todos os registros afetados serão utilizados no agrupamento solicitado ou se serão descartadas possíveis duplicidades, respectivamente.

Outro ponto importante a se considerar é quanto ao conteúdo armazenado no atributo da expressão. O registro que tiver o conteúdo NULL no atributo da expressão não será contabilizado no agrupamento.

É sempre útil consolidar novos conhecimentos através de exemplos. Assim, para ilustrar o uso de funções agregadas, vamos tomar como base o conjunto de tabelas apresentado na figura 6.1.



codalu [PK] serial	nomealu character varying(30)
1	Ana Maria Silveira
2	Beatriz Killing
3	Carlos Teixeira
4	Daniel Silva
5	Elizabete Torres
6	Flavio Ribeiro
7	Gabriel Pensador
8	Igor Camargo

codmat [PK] serial	codalu integer	coddisci integer	nota numeric(4,1)	faltas integer
1	1	1	7.5	0
2	1	3	8.2	0
3	1	5	6.3	0
4	2	1	4.7	4
5	2	3	5.8	2
6	2	5	7.3	2
7	3	2	8.5	0
8	3	4	9.0	0
9	3	6	10.0	0
10	4	2	7.0	0
11	4	4	5.0	0
12	5	1	4.0	8
13	5	5	4.0	2
14	6	4	3.0	0
15	6	6	0.0	8
16	7	7	8.0	2
17	7	8	5.0	8

coddisci [PK] serial	nomedisci character varyi	codprof integer
1	Linguagem 1	1
2	Linguagem 2	1
3	Sistemas 1	2
4	Sistemas 2	2
5	Redes 1	3
6	Redes 2	3
7	Portugues	4
8	Matematica	5

codprof [PK] serial	nomeprof character varying(30
1	JOAO DA SILVA
2	MARIA SANTOS
3	RICARDO SOUZA
4	PAULO RODRIGUES
5	SANDRA TEIXEIRA

Figura 6.1
Modelo de dados 'Escola' utilizado para ilustrar o uso de funções agregadas.

A seguir, na figura 6.2, apresentamos um conjunto de exemplos onde as funções agregadas são usadas.

Exemplo 1: contar o número de alunos com nota inferior a 7, em qualquer disciplina.

```
SELECT COUNT(DISTINCT codalu)
FROM matricula
WHERE nota < 7;
```

pane

a Output | Explain | Messages | His

count bigint
6

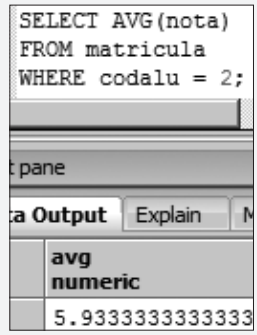
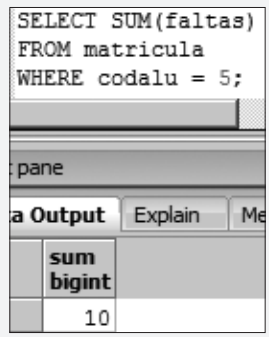
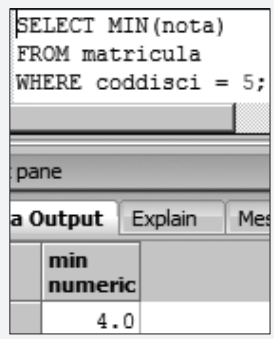
Exemplo 2: média das notas do aluno com código 2.	
Exemplo 3: total de faltas do aluno com código 5.	
Exemplo 4: menor nota da disciplina 5 (Redes I).	

Figura 6.2
Exemplos do uso de funções agregadas.

Agrupamento

Agrupamento é um recurso que podemos utilizar no processamento de consultas SQL/DML quando há a necessidade de trabalharmos com funções agregadas em conjuntos específicos de dados do banco, e não sobre todos os dados.

Em muitos casos é necessário aplicar as funções agregadas para um subgrupo de tuplas definido com base em alguns atributos. Por exemplo, considere a necessidade de calcular a média de salário de cada departamento ou número de empregados por projeto.

Isso é possível através do uso da cláusula GROUP BY, que permite particionar a relação em grupos. Esses grupos não podem se sobrepor, tomando como base valores iguais dentro de um ou mais de seus atributos (atributo de agrupamento).

Para demonstrar o uso de agrupamento, vamos utilizar um outro modelo de dados, reproduzindo informações sobre uma empresa qualquer e que pode ser visto na figura 6.4.

Empregado

ssn	pnome	inicialm	unome	sexo	endereco	datanasc	superssn	dno	salario
123456789	Joao	B	Souza	M	R. 24 de maio, 1500 - Curitiba - PR	1965-01-09	333445555	5	30000
333445555	Fabio	T	Will	M	R. Alagoas, 325 - Curitiba - PR	1955-12-08	888665555	5	40000
999887777	Alice	J	Zebra	F	R. Ribeira, 98 - Pinhais - PR	1968-01-19	987654321	4	25000
987654321	Jennifer	S	Wallace	F	R. Berry, 291 - Colombo - PR	1941-06-20	888665555	4	43000
666884444	Ricardo	K	Nantes	M	Av. Figueira, 55 - Almirante Tamandare - PR	1962-09-15	333445555	5	38000
453453453	Jussara	A	Pereira	F	Trav. da Lapa, 23 - Curitiba - PR	1972-07-31	333445555	5	25000
987987987	Alberto	V	Medeiros	M	R. Ceará, 1245 - Curitiba - PR	1969-03-29	987654321	4	25000
888665555	Joaquim	E	Brito	M	R. Stone, 450 - Curitiba - PR	1937-11-10	(Null)	1	55000

Localização

dnum	dlocalizacao
1	Curitiba
4	Colombo
5	Araucaria
5	Curitiba
5	Pinhais

Dependente

essn	nomedep	sexodep	datanascdep	parentesco
333445555	Alice	F	1986-04-05	FILHA
333445555	Teodoro	M	1983-10-25	FILHO
333445555	Joana	F	1958-05-03	CONJUGE
987654321	Abdala	M	1942-02-28	CONJUGE
123456789	Michel	M	1988-01-04	FILHO
123456789	Alice	F	1988-12-30	FILHA
123456789	Elizabete	F	1967-05-05	CONJUGE

Trabalha

essn	pno	horas
123456789	1	32,5
123456789	2	7,5
666884444	3	40
453453453	1	20
453453453	2	20
333445555	2	10
333445555	3	10
333445555	10	10
333445555	20	10
999887777	30	30
999887777	10	10
987987987	10	35
987987987	30	5
987654321	30	20
987654321	20	15
888665555	20	(Null)

Departamento

dnumero	dnome	gerssn	gerdatainicio
5	Pesquisa	333445555	1988-05-22
4	Administracao	987654321	1995-01-01
1	Sede Administrativa	888665555	1981-06-19

Projeto

pnumero	pjnome	plocal	dnum
1	ProdutoX	Pinhais	5
2	ProdutoY	Araucaria	5
3	ProdutoZ	Curitiba	5
10	Automatizacao	Pinhais	4
20	Reorganizacao	Curitiba	1
30	Novos Beneficios	Pinhais	4

Figura 6.3
DB Empresa – base para diversos exemplos nesta sessão.

Exemplo 1: para cada departamento, listar seu nome, número de empregados que nele trabalham e a média de seus salários.

Importante: o PostgreSQL exige a indicação dos campos selecionados (que não aplicados no grupo ou funções agregadas) na cláusula GROUP BY: e.dno, **d.dnome**;

```
SELECT d.dnome, COUNT(*), AVG(e.salario)
FROM empregado e
INNER JOIN departamento d ON e.dno = d.dnumero
GROUP BY e.dno, d.dnome;
```

pane

Output | Explain | Messages | History

dnome character varying(30)	count bigint	avg double p
Pesquisa	4	33250
Administracao	3	31000
Sede Administrativa	1	55000

Exemplo 2: para cada projeto, liste seu número, seu nome e o número de empregados que nele trabalham.

```
SELECT p.pnumero, p.pjnome, COUNT(*)
FROM trabalha t
INNER JOIN projeto p ON t.pno = p.pnumero
GROUP BY t.pno, p.pnumero, p.pjnome;
```

pane

Output | Explain | Messages | History

pnumero character(2)	pjnome character varying(30)	count bigint
2	ProdutoY	3
3	ProdutoZ	2
1	ProdutoX	2
20	Reorganizacao	3
10	Automatizacao	3
30	Novos Beneficios	3

Figura 6.4
Exemplos do uso de agrupamento.

Algumas vezes é necessário recuperar os valores dessas funções agregadas somente para grupos que satisfaçam certas condições. Isso pode ser conseguido através do argumento HAVING, similar ao parâmetro WHERE. Sua função é a de filtrar registros dentro de uma consulta SQL/DML em execução. A diferença é que o HAVING é aplicado sobre os registros agrupados pelo argumento GROUP BY.

A cláusula GROUP BY deve ser colocada antes da HAVING, pois os grupos são formados e as funções agregadas são calculadas antes de se resolver a cláusula HAVING. Ilustramos o uso dessas cláusulas nos exemplos da figura 6.5.

Exemplo 3: igual ao Exemplo 2, exceto por considerar apenas projetos com mais de dois empregados.

```
SELECT p.pnumero, p.pjnome, COUNT(*) AS "No. Emp"
FROM trabalha t
INNER JOIN projeto p ON t.pno = p.pnumero
GROUP BY t.pno, p.pnumero, p.pjnome
HAVING COUNT(*) > 2;
```

pnumero character(2)	pjnome character varying(30)	No. Emp bigint
2	ProdutoY	3
20	Reorganizacao	3
10	Automatizacao	3
30	Novos Beneficios	3

Exemplo 4: a cláusula WHERE não pode ser utilizada para restringir grupos que deverão ser exibidos.

```
SELECT d.dnome, AVG(e.salario)
FROM departamento d
INNER JOIN empregado e ON e.dno = d.dnumero
WHERE COUNT(e.ssn) > 2
GROUP BY e.dno, d.dnome;
```

Resultado:
ERRO: agregação não é permitida na cláusula WHERE.
LINE 4: WHERE COUNT(e.ssn) > 2

Exemplo 5: listar o nome da disciplina e a média de notas em cada uma delas, para disciplinas com mais de doi alunos matriculados.

```
SELECT d.nomedisci, AVG(m.nota)
FROM disciplina d
INNER JOIN matricula m ON m.coddisci = d.coddisci
GROUP BY m.coddisci, d.nomedisci
HAVING COUNT(codmat) > 2;
```

nomedisci character varyi	avg numeric
Sistemas 2	5.6666666666666667
Redes 1	5.8666666666666667
Linguagem 1	5.4000000000000000

Figura 6.5
Exemplos do uso da cláusula HAVING.

Lembre-se, portanto, de que a cláusula HAVING proporciona a aplicação de uma condição para o grupo de tuplas associado a cada valor dos atributos do agrupamento. Seu funcionamento está diretamente ligado ao uso da cláusula GROUP BY e não deve ser confundido com a cláusula WHERE.

Funções Nativas

Sistemas Gerenciados de Banco de Dados geralmente fornecem um conjunto de funções que auxiliam na manipulação dos tipos de dados primitivos por eles disponibilizados.

A grande maioria dessas funcionalidades é dependente de um determinado SGDB, não sendo especificadas pelo padrão SQL. Operadores para cadeia de caracteres, funções para formatação dos tipos data e hora, funções e operações geométricas, entre outros, são recursos que os SGBDs podem oferecer para seus usuários com o intuito de facilitar a realização de determinadas operações sobre os dados neles armazenados.

Informações do sistema e de sessão

Obter versão do SGBD, data/hora do sistema, usuário e database atuais.

```
SELECT version(), now();

SELECT current_user, current_database();
```

version text	now timestamp with time zone	current_user name	current_database name
PostgreSQL 9.3.1,	2014-04-09 15:36:10.714-03	postgres	empresa

Figura 6.6
Funções Nativas – Informações de SGBD e sessão.

Manipulação de data/hora

Obter data e hora atuais e obter partes do timestamp (Padrão SQL-2003).

```
SELECT current_date, current_time, localtimestamp;
```

date date	timetz time with time zone	timestamp timestamp without time zone
2014-04-11	14:34:23.822-03	2014-04-11 14:34:23.822

```
EXTRACT(field FROM timestamp)
```

Onde **field** pode ser qualquer um dos seguintes identificadores:

century	milliseconds
day	minute
decade	Month
dow / isodow: day of the week as Sunday(0/1) to Saturday(6/7)	quarter second
doy: day of the year (1: 365/366)	timezone
epoch	timezone_hour
hour	timezone_minute
microseconds	week
millennium	year / isoyear

Figura 6.7
Funções que manipulam data, hora e timestamp.

Um exemplo interessante, e muitas vezes útil, é o de como fazer para calcular idades. Na figura a seguir, são exibidos dois métodos diferentes.

Método 1: uso de operador para calcular a diferença de anos.

```
SELECT pnome,
    EXTRACT(YEAR FROM NOW()) –
    EXTRACT(YEAR FROM datanasc)
    AS idade
    FROM empregado;
```

pnome character	idade double
Joao	49
Fabio	59
Alice	46
Jennifer	73
Ricardo	52
Jussara	42
Alberto	45
Joaquim	77

Método 2: função AGE

```
SELECT pnome, AGE(datanasc) AS idade
FROM empregado;
```

pnome character	idade interval
Joao	49 years 3 mons
Fabio	58 years 4 mons 1 day
Alice	46 years 2 mons 21 days
Jennifer	72 years 9 mons 19 days
Ricardo	51 years 6 mons 24 days
Jussara	41 years 8 mons 9 days
Alberto	45 years 11 days
Joaquim	76 years 4 mons 29 days

Figura 6.8
Métodos para cálculo de idade.

Outra função nativa para manipular datas é a DATE_PART, semelhante à função EXTRACT vista acima e que inclusive utiliza os mesmos identificadores apresentados na figura 6.7 como parâmetros. Sua sintaxe é a seguinte:

```
DATE_PART(text, timestamp)
```

No exemplo a seguir as são utilizadas as funções EXTRACT e DATE_PART para extrair partes da data atual, NOW(), com o resultado obtido sendo apresentado logo a seguir.

```
SELECT EXTRACT(CENTURY FROM NOW()) as seculo,

   DATE_PART('doy', NOW()) as diasano, DATE_PART('day', NOW()) as
diaatual,

   DATE_PART('hour', NOW()) as hora, DATE_PART('month', NOW()) as
mesatual;
```

seculo double precision	diasano double precision	diaatual double precision	hora double precision	mesatual double precision
21	105	15	21	4

Na figura 6.9, temos uma série de exemplos ara ilustrar o uso de operadores com datas.

calcular nova data a partir de um intervalo específico:

```
SELECT date '2014-02-10' AS compra,
       date '2014-02-10' + 30 AS vencimento;
```

compra date	vencimento date
2014-02-10	2014-03-12

calcular intervalo de dias entre datas.

```
SELECT date '2014-03-28': date '2014-03-12' as diferenca;
```

	diferenca integer
1	16

```
SELECT time '05:00:30': time '03:00:46';
```

?column? interval
01:59:44

```
SELECT timestamp '2001-09-28 23:00' - interval '28 hours';
```

?column? timestamp without time zone
2001-09-27 19:00:00

```
SELECT 500 * interval '1 second';
```

?column? interval
00:08:20

Figura 6.9
Funções Nativas – exemplo de uso de operador com data e hora.

As funções para formatação de Data/Hora são também muito utilizadas, onde TO_CHAR converte um carimbo do tempo em um conjunto de caracteres, e TO_DATE converte um conjunto de caracteres em data. A sintaxe de TO_CHAR é:

```
TO_CHAR(timestamp, text)
```

Vejam que text é um conjunto de caracteres que pode funcionar como uma máscara para um formato final desejado. É o caso da máscara 'DD/MM/YYYY', onde DD será substituído por dia do mês. Qualquer parte do texto que não seja reconhecido como um modelo ou máscara padrão é simplesmente copiado sem alteração.

Existe uma grande quantidade de modelos ou máscaras padrão para a formatação de data e hora.

É importante lembrar que a função TO_CHAR pode converter outros tipos de dados em um conjunto de caracteres, bastando passar como parâmetro um número (int ou double) no lugar do timestamp. Nesse caso, haverá modelos ou máscaras padrão para números, também disponíveis nesse link.

A seguir, apresentamos exemplos de formatação de datas.

```
SELECT TO_CHAR(now(), 'DD/MM/YYYY');
```

to_char text
11/04/2014

```
SELECT TO_CHAR(now(), 'FMday, DD FMmonth YYYY');
```

to_char text
friday, 11 april 2014

```
SELECT TO_CHAR(current_timestamp, 'HH24:MI:SS');
```

to_char text
15:05:10

Figura 6.10
Funções Nativas – exemplo de formatação de data.

Manipulação de String

A manipulação de strings pode ser feita através de operadores e funções nativas. O operador de concatenação || vai juntando conjuntos de caracteres em sequência e as funções existentes permitem extrair ou substituir trechos de strings, contar o número de caracteres, alterar a caixa, entre outras funções. Na figura a seguir apresentamos alguns exemplos.

Concatenação: operador ||.

```
SELECT pnome || ' ' || inicialm || '. ' || unome
       AS “Nome Completo” FROM empregado;
```

Nome Completo text
Joao B. Souza
Fabio T. Will
Alice J. Zebra
Jennifer S. Wallace
Ricardo K. Nantes
Jussara A. Pereira
Alberto V. Medeiros
Joaquim E. Brito

Conversão caixa alta–UPPER e caixa baixa-LOWER.

```
SELECT UPPER('banco'), LOWER('BANCO');
```

upper text	lower text
BANCO	banco

Obter número de caracteres em uma string (length).

```
SELECT CHAR_LENGTH('banco');
```

char_length integer
5

Converte a 1ª letra de cada palavra em caixa alta.

```
SELECT INITCAP('oi MUNDO');
```

initcap text
Oi Mundo

Figura 6.11
Funções Nativas – exemplo de formatação de string.

Já o exemplo da figura 6.12 aplica um conjunto de funções para o nome dos dependentes na tabela dependente da figura 6.3. As funções utilizadas são:

- **POSITION**: obter a localização de uma substring;
- **SUBSTRING**: obter a substring a partir de uma posição, x caracteres;
- **OVERLAY**: substituir uma substring a partir de uma posição, x caracteres.

```
SELECT nomedep,
   POSITION('ana' IN nomedep),
   SUBSTRING(nomedep FROM 3 FOR 4),
   SUBSTRING(nomedep,3,4),
   OVERLAY(nomedep PLACING 'pos' FROM 2 FOR 4)
FROM dependente;
```

nomedep character v.	position integer	substring text	substring text	overlay text
Alice	0	ice	ice	Apos
Teodoro	0	odor	odor	Tposro
Joana	3	ana	ana	Jpos
Abdala	0	dala	dala	Aposa
Michel	0	chel	chel	Mposl
Alice	0	ice	ice	Apos
Elizabete	0	izab	izab	Eposbete

Figura 6.12
Funções Nativas – exemplo de manipulação de strings.

Manipulação de números

O PostgreSQL apresenta uma quantidade considerável de operadores matemáticos que podem ser utilizados em expressões numéricas com os tipos de dados suportados. Os operadores bit a bit trabalham somente em tipos de dado inteiros, enquanto os demais estão disponíveis para todos os tipos de dado numéricos.

A figura 6.13 traz uma tabela com os operadores matemáticos disponíveis e exemplos de uso para cada um deles. A figura traz ainda alguns exemplos adicionais.

Operador	Descrição	Exemplo	Resultado
+	adição	2 + 3	5
–	subtração	2 – 3	1
*	multiplicação	2 * 3	6
/	divisão (divisão inteira trunca o resultado	04/fev	2
%	módulo (resto)	5 % 4	1
^	exponenciação	2.0 ^ 3.0	8
|/	raiz quadrada	|/ 27.0	5
|//	raiz cúbica	|// 27.0	3
!	fatorial	5 !	120
!!	fatorial (operador de prefixo)	!! 5	120
@	Valor absoluto	@ - 5.0	5
&	AND bit a bit	91 & 15	11
|	OR bit a bit	32 | 3	35
#	XOR bit a bit	17 # 5	20
~	NOT bit a bit	~ 1	-2
<<	deslocamento à esquerda bit a bit	1 << 4	16
>>	deslocamento à direita bit a bit	8 >> 4	2

Exemplos:	Resultados:
5! AS fat, 2^3 AS pot, 5%3 AS resto, |/9 AS raiz2, @ -5.2 AS abs;	fat num / pot doub / resto intege / raiz2 double / abs num 120 / 8 / 2 / 3 / 5.2

Figura 6.13
Funções Nativas – operadores matemáticos.

É grande também a quantidade de funções disponíveis para manipulação de números, incluindo funções matemáticas, trigonométricas e geométricas. Apresentamos a seguir, na figura 6.14, algumas dessas funções com exemplos de uso e respectivos resultados.

Função	Descrição	Exemplo	Resultado
abs()	valor absoluto	abs(-17.4)	17.4
cbrt()	raiz cúbica	cbrt(27.0)	3
ceil()	o menor inteiro não menor que o argumento	ceil(-42.8)	-42
exp()	exponenciação	exp(1.0)	2.71828182845905
floor()	o maior inteiro não maior que o argumento	floor(-42.8)	-43
log()	logaritmo na base 10	log(100.0)	2
mod(x,y)	resto de y/x	mod(9,4)	1
pi()	constante "π"	pi()	3.14159265358979
power(a,b)	a elevado a b	power(9.0, 3.0)	729
random()	valor randômico entre 0.0 e 1.0	random()	
round()	arredondar para o inteiro mais próximo	round(42.4)	42
round(v,s)	arredondar para s casas decimais	round(42.4382, 2)	42.44
setseed()	define a semente para as chamadas a random()	setseed(0.54823)	1177314959
sqrt()	raiz quadrada	sqrt(2.0)	1.4142135623731
trunc()	trunca em direção ao zero	trunc(42.8)	42
trunc(v,s)	trunca com s casas decimais	trunc(42.4382, 2)	42.43

Figura 6.14
Algumas das funções matemáticas e trigonométricas.

Outras funções

A quantidade de funções nativas disponíveis no PostgreSQL é realmente muito grande e variada. Para os que tiverem interesse, sugerimos consultar a documentação disponível na internet. De qualquer modo, apresentamos mais alguns exemplos de funções.

ASCII(<char>): Retorna código ASCII

```
SELECT ASCII('A');
```

ascii integer	chr text
65	B

CHR(<intCodASCII>): Retorna o caractere

```
SELECT CHR(66);
```

LEFT(<str>,<n>): Retorna os n primeiros caracteres

```
SELECT LEFT('morango',4);
```

RIGHT(<str>,<n>): Retorna os n últimos caracteres

```
SELECT RIGHT('morango',4);
```

right text	left text
ango	mora

REVERSE(<str>): Retorna uma string invertida

```
SELECT REVERSE('banco de dados');
```

reverse text
sodad ed ocnab

ENCODE(<text>,<formato>)

```
SELECT ENCODE('123', 'base64'),
```

DECODE(<bytea>,<formato>)

```
SELECT DECODE('MTIz','base64');
```

encode text	decode bytea
MTIz	123

Figura 6.15
Outros exemplos de funções nativas.

Nesses exemplos, o parâmetro <formato> das funções ENCODE() e DECODE(), respectivamente para codificação e decodificação de caracteres, pode assumir os seguintes valores: base64, hex ou escape.

Atividade de Fixação

Liste duas funções agregadas ou nativas que não foram tratadas nessa sessão mas que estejam listadas no link indicado no AVA.

7

Subconsultas, índices e visões

objetivos

Entender o processo de escrita e utilização de subconsultas, índices e visões, em conformidade com o padrão SQL implementado no PostgreSQL.

conceitos

Subconsultas; Índices, table scan; explain; query plan; Visões.

Exercício de nivelamento

Descreva com suas palavras o que você entende por subconsultas, índices e visões dentro de um banco de dados.

Subconsultas

- Existem três tipos de subconsultas:
- ESCALAR: retorna um único valor.
- ÚNICA LINHA: retorna uma lista de valores.
- TABELA: retorna uma ou mais colunas e múltiplas linhas.

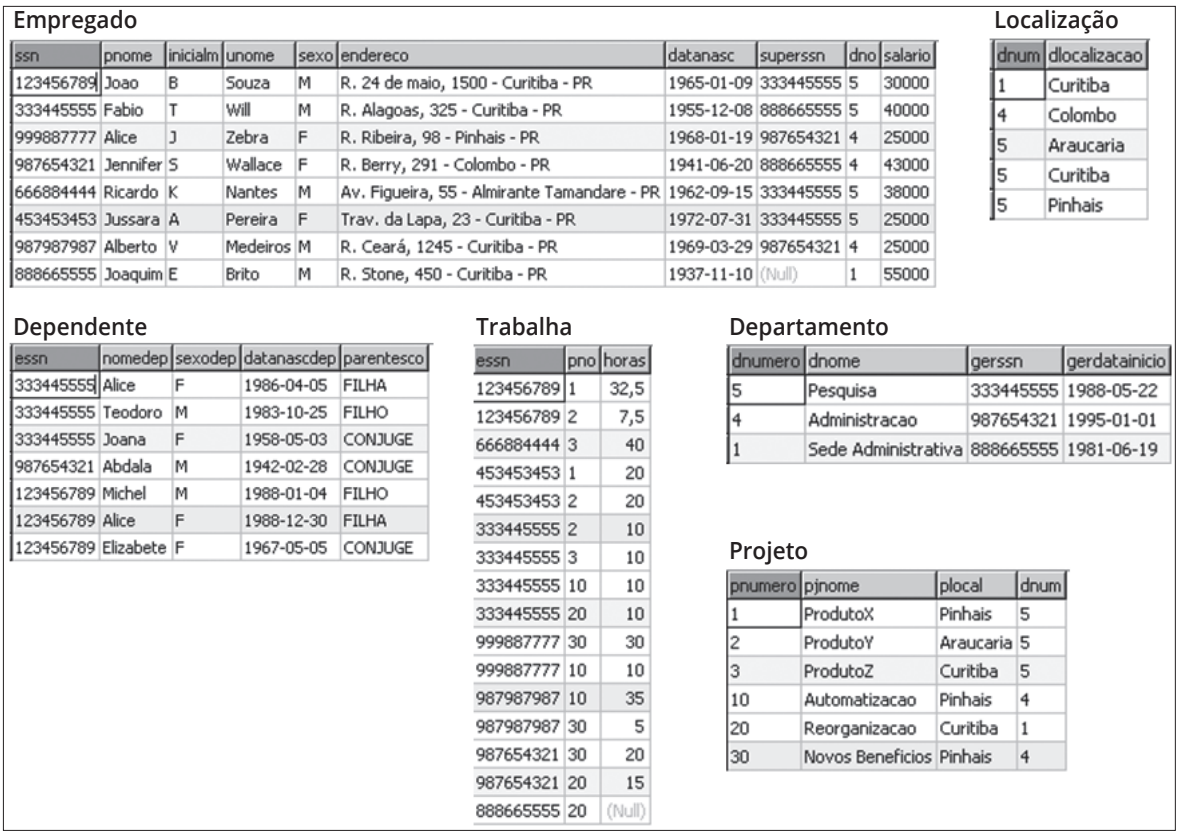
Podemos definir uma subconsulta como a combinação de uma consulta (interna) dentro de outra consulta (externa). Esse recurso possibilita que o resultado da consulta mais interna seja utilizado pela consulta mais externa.

Como foi visto na sessão anterior, funções agregadas são utilizadas para produzir resultados únicos para um conjunto de dados contendo várias linhas (tuplas) fornecidas como entrada, geralmente representando dados armazenados em tabelas.

Já a subconsulta, ou consulta mais interna, pode ser usada no SELECT como uma coluna de projeção ou nas cláusulas FROM, WHERE, GROUP BY e HAVING.

É importante lembrar que as subconsultas devem ser sempre escritas entre parênteses: '(q)'.

Nesta sessão serão apresentados diversos exemplos para ajudar na compreensão dos conceitos aqui apresentados. Em todos esses exemplos, serão utilizadas tabelas e dados conforme exibidos no modelo que aparece na figura 7.1.



Empregado

ssn	pnome	inicialm	unome	sexo	endereco	datanasc	superssn	dno	salario
123456789	Joao	B	Souza	M	R. 24 de maio, 1500 - Curitiba - PR	1965-01-09	333445555	5	30000
333445555	Fabio	T	Will	M	R. Alagoas, 325 - Curitiba - PR	1955-12-08	888665555	5	40000
999887777	Alice	J	Zebra	F	R. Ribeira, 98 - Pinhais - PR	1968-01-19	987654321	4	25000
987654321	Jennifer	S	Wallace	F	R. Berry, 291 - Colombo - PR	1941-06-20	888665555	4	43000
666884444	Ricardo	K	Nantes	M	Av. Figueira, 55 - Almirante Tamandare - PR	1962-09-15	333445555	5	38000
453453453	Jussara	A	Pereira	F	Trav. da Lapa, 23 - Curitiba - PR	1972-07-31	333445555	5	25000
987987987	Alberto	V	Medeiros	M	R. Ceará, 1245 - Curitiba - PR	1969-03-29	987654321	4	25000
888665555	Joaquim	E	Brito	M	R. Stone, 450 - Curitiba - PR	1937-11-10	(Null)	1	55000

Localização

dnum	dlocalizacao
1	Curitiba
4	Colombo
5	Araucaria
5	Curitiba
5	Pinhais

Dependente

essn	nomedep	sexodep	datanascdep	parentesco
333445555	Alice	F	1986-04-05	FILHA
333445555	Teodoro	M	1983-10-25	FILHO
333445555	Joana	F	1958-05-03	CONJUGE
987654321	Abdala	M	1942-02-28	CONJUGE
123456789	Michel	M	1988-01-04	FILHO
123456789	Alice	F	1988-12-30	FILHA
123456789	Elizabete	F	1967-05-05	CONJUGE

Trabalha

essn	pno	horas
123456789	1	32,5
123456789	2	7,5
666884444	3	40
453453453	1	20
453453453	2	20
333445555	2	10
333445555	3	10
333445555	10	10
333445555	20	10
999887777	30	30
999887777	10	10
987987987	10	35
987987987	30	5
987654321	30	20
987654321	20	15
888665555	20	(Null)

Departamento

dnumero	dnome	gerssn	gerdatainicio
5	Pesquisa	333445555	1988-05-22
4	Administracao	987654321	1995-01-01
1	Sede Administrativa	888665555	1981-06-19

Projeto

pnumero	pjnome	plocal	dnum
1	ProdutoX	Pinhais	5
2	ProdutoY	Araucaria	5
3	ProdutoZ	Curitiba	5
10	Automatizacao	Pinhais	4
20	Reorganizacao	Curitiba	1
30	Novos Beneficios	Pinhais	4

Figura 7.1
Modelo de dados de uma Empresa utilizado como referencia nos exemplos.

A seguir, analisaremos os três tipos de subconsultas acima mencionados.

Subconsulta Escalar

É uma consulta interna que retorna exatamente um único valor, ou seja, uma única linha com uma única coluna. Esse valor será utilizado para que a consulta mais externa produza o resultado esperado.

Subconsultas escalares são aceitáveis (e muitas vezes muito úteis) em praticamente qualquer situação onde você poderia usar um valor literal, uma constante ou uma expressão para filtrar registros durante o processo de consulta.

A figura 7.2 trás um primeiro exemplo com o uso de operadores. Nele são listados os empregados que trabalham no departamento de Pesquisa.

```
SELECT ssn, pnome, salario
FROM empregado
WHERE dno = (SELECT dnumero
             FROM departamento
             WHERE dnome = 'Pesquisa');
```

ssn character(9)	pnome character varying(30)	salario double precision
123456789	Joao	30000
333445555	Fabio	40000
666884444	Ricardo	38000
453453453	Jussara	25000

Figura 7.2 Subconsulta escalar – uso de operadores.

Já o exemplo apresentado na figura 7.3 faz uso de função agregada. Nele são listados os empregados cujos salários são maiores do que o salário médio, mostrando o quanto são maiores (a diferença).

```
SELECT ssn, pnome, salario - (SELECT AVG(salario)
                              FROM empregado) AS difSal
FROM empregado
WHERE salario > (SELECT AVG(salario)
                 FROM empregado);
```

ssn character(9)	pnome character varying(30)	difsal double precision
333445555	Fabio	4875
987654321	Jennifer	7875
666884444	Ricardo	2875
888665555	Joaquim	19875

Média: 35125

Figura 7.3 Subconsulta escalar – uso de função agregada.

Por fim, temos um exemplo onde se faz uso de função agregada. Na figura 7.4 é apresentada a consulta e seu resultado, listando o nome do empregado e do departamento onde o empregado mais novo do sexo masculino está alocado.

```
SELECT e.pnome, d.dnome, AGE(e.datanasc)
FROM empregado e
INNER JOIN departamento d ON d.dnumero = e.dno
WHERE AGE(e.datanasc) = (SELECT MIN(age(e.datanasc))
                         FROM empregado e
                         WHERE e.sexo = 'M');
```

pnome character varying(30)	dnome character varying(30)	age interval
Alberto	Administracao	45 yea

Figura 7.4 Subconsulta escalar – uso de função agregada.

Subconsulta ÚNICA LINHA

Uma subconsulta do tipo ÚNICA LINHA é aquela em que a consulta interna retorna uma lista de valores, sendo esses valores utilizados na consulta mais externa que vai produzir o resultado esperado.

Nesse tipo de subconsulta é comum utilizar cláusulas para delimitar o escopo da consulta, conforme a tabela apresentada na figura 7.5.

SÍMBOLO	SIGNIFICADO
IN	Igual a qualquer um dos valores da lista.
NOT IN	Diferente dos valores existente na lista.
ANY/SOME	Retorna linhas que correspondam a qualquer um dos valores existente na lista.
ALL	Retorna linhas que correspondam a todos os valores existentes na lista.

Figura 7.5
Subconsulta Única Linha – cláusula IN.

Vejamos exemplos do uso de cada uma das cláusulas apresentadas.

Cláusulas IN

São usadas em subconsultas que produzem resultado tendo como base os valores retornados na subconsulta mais interna.

Saiba mais

A tradução de IN é EM, ou seja, "que faz parte".

No exemplo da figura 7.6, temos uma consulta que lista os dependentes dos funcionários que trabalham no departamento de Pesquisa.

```
SELECT nomedep, datanascdep, parentesco
FROM dependente
WHERE essn IN (SELECT ssn
           FROM empregado
           WHERE dno = (SELECT dnumero
                        FROM departamento
                        WHERE dnome = 'Pesquisa'));
```

nomedep character v	datanascdep date	parentesco character v
Michel	1988-01-04	FILHO
Alice	1988-12-30	FILHA
Elizabete	1967-05-05	CONJUGE
Alice	1986-04-05	FILHA
Teodoro	1983-10-25	FILHO
Joana	1958-05-03	CONJUGE

Figura 7.6
Subconsulta Única Linha – cláusula IN.

Cláusulas NOT IN

São usadas em subconsultas que produzem resultado tendo como base os valores que são diferentes dos retornados na subconsulta mais interna. A tradução de NOT IN é NÃO EM, ou seja, que não faz parte (ou não está contido em).

O exemplo de consulta que utiliza a cláusula NOT IN pode ser visto na figura 7.7 e mostrar o nome do empregado que *não possui* dependente.

```
SELECT pnome, unome

FROM empregado

WHERE ssn NOT IN (SELECT essn

                  FROM  dependente);
```

Empregados e seus dependentes

essn character(9)	pnome character	nomedep character va
333445555	Fabio	Alice
333445555	Fabio	Teodoro
333445555	Fabio	Joana
987654321	Jennifer	Abdala
123456789	Joao	Michel
123456789	Joao	Alice
123456789	Joao	Elizabete

Empregados sem dependentes

pnome character varying(30)	unome character varying(30)
Alberto	Medeiros
Alice	Zebra
Joaquim	Brito
Jussara	Pereira
Ricardo	Nantes

Figura 7.7
Subconsulta Única Linha – cláusula NOT IN.

Vejam que o resultado no exemplo ao lado é composto, unicamente, pelas colunas selecionadas no SELECT mais externo.

Um segundo exemplo do uso de NOT IN é apresentado a seguir. Nele é listado o nome das pessoas da instituição que sejam apenas professores, demonstrando como trabalhar com generalização ou especialização não exclusiva (ou compartilhada). Nesse caso, imagine a existência da entidade pessoa (generalizada) e suas especializações professor e aluno.

```
SELECT nome

FROM professor NOT IN (SELECT nome FROM aluno);
```

Cláusulas ANY/SOME

São usadas com subconsultas (consulta mais interna) que produzem um conjunto de valores numéricos (ÚNICA LINHA, semelhante ao IN). O resultado final, consulta mais externa, é obtido com base em um dos valores retornados pela consulta mais interna. A tradução de ANY/SOME é ALGUM.

O uso de operadores com a cláusula SOME pode gerar algumas confusões. Estejam atentos para o seguinte:

- = some: idêntico ao IN
- > some: maior que algum (outros >=, <=, <,<>)
- <> some: não é o mesmo que NOT IN

Na figura 7.8 aparece um exemplo de uso da cláusula SOME onde são listados os empregados cujos salários são maiores do que o salário de pelo menos um (algum) funcionário do departamento 5-Pesquisa.

```
SELECT pnome, salario
FROM empregado
WHERE salario > SOME (SELECT salario
                      FROM empregado
                      WHERE dno = '5');
```

pnome character varying(30)	salario real
Joaquim	55000
Fabio	40000
Jennifer	43000
Joao	30000
Ricardo	38000

Figura 7.8
Subconsulta Única Linha – cláusula ANY/SOME.

Cláusula ALL

Utilizada com subconsultas (consulta mais interna) que produzem um conjunto de valores numéricos. O resultado final, consulta mais externa, é obtido com base todos dos valores retornados pela consulta mais interna. A tradução de ALL é TODOS, ou seja, a comparação é feita em relação a todos os elementos contidos no resultado da subconsulta.

Assim, vejamos algumas equivalências no uso de operadores com o ALL:

- > ALL: maior que todos (outros >=, <=, <,<>)
- <> ALL: é o mesmo que NOT IN

No exemplo, na figura 7.9, temos o uso do > ALL, que traz como resultado a lista dos empregados cujos salários são maiores do que o salário de cada funcionário do departamento 5-Pesquisa.

```
SELECT pnome, salario
FROM empregado
WHERE salario > ALL (SELECT salario
                     FROM empregado
                     WHERE dno = '5');
```

pnome character varying(30)	salario real
Joaquim	55000
Jennifer	43000

Figura 7.9
Subconsulta Única Linha – cláusula ALL.

Consultas aninhadas correlacionadas

Nas consultas aninhadas correlacionadas, a consulta mais interna precisa de um dado que vem da consulta mais externa (query principal). Isso faz com que o SELECT interno seja executado tantas vezes quantas forem necessárias, dependendo diretamente do número de linhas que são processadas na query principal.

Essa situação pode ser melhor entendida através de um exemplo, conforme o apresentado na figura 7.10, onde deseja-se consultar o nome de empregado(s) que tenha(m) dependente(s) cujo sexo seja o mesmo do empregado em questão.

```
SELECT pnome, unome
FROM empregado e
WHERE e.ssn IN (SELECT d.essn
                          FROM dependente d
                          WHERE e.sexo = d.sexodep);
```

pnome character varying(30)	unome character varying(30)
Fabio	Will
Joao	Souza

Figura 7.10
Subconsulta Única Linha – cláusula ALL.

Uma alternativa para essa consulta é fazer uso de um INNER JOIN conforme mostrado a seguir. Destacamos, inclusive, que o INNER JOIN garante um melhor desempenho sempre que comparado com consultas aninhadas correlacionadas.

```
SELECT pnome, unome FROM empregado e
INNER JOIN dependente d ON e.ssn = d.essn WHERE e.sexo = d.sexodep
```

Subconsulta TABELA

É uma consulta interna que retorna uma ou mais colunas e múltiplas linhas, ou seja, apresenta como resultado uma tabela temporária. Nesse caso, a subconsulta faz referência às variáveis da consulta que a envolve, vistas como constantes durante a execução da subconsulta.

As cláusulas EXISTS e NOT EXISTS foram projetadas para uso apenas com subconsultas do tipo TABELA, indicando se o resultado de uma pesquisa contém ou não tuplas. No exemplo da figura 7.11, vemos o uso da cláusula EXISTS na consulta aos empregados que trabalham no departamento de Pesquisa. A figura mostra ainda o INNER JOIN equivalente.

```
SELECT pnome, unome FROM empregado e
WHERE EXISTS (SELECT d.dnumero FROM departamento d
      WHERE e.dno = d.dnumero AND d.dnome = 'Pesquisa');
```

pnome characte	unome characte
Joao	Souza
Fabio	Will
Ricardo	Nantes
Jussara	Pereira

Figura 7.11
Subconsulta TABELA – cláusula EXISTS.

Melhor desempenho:

```
SELECT pnome, unome FROM empregado e
INNER JOIN departamento d ON d.dnumero = e.dno
                WHERE d.dnome = 'Pesquisa';
```

Já no exemplo da figura 7.12, temos o uso simultâneo das cláusulas EXISTS e NOT EXISTS, esta última permitindo descobrir quem são os funcionários que não possuem dependentes.

```
SELECT pnome, unome
FROM empregado e
WHERE EXISTS (SELECT *
      FROM departamento dp
      WHERE e.ssn = dp.gerssn)
   AND NOT EXISTS
     (SELECT *
      FROM dependente d
            WHERE e.ssn = d.essn);
```

pnome character varying(30)	unome character varying(30)
Joaquim	Brito

Figura 7.12
Subconsulta TABELA – cláusula NOT EXISTS.

Inserir dados recuperados de uma tabela em outra (uso do SELECT)

É interessante saber que é possível fazer a inclusão de registros resultantes de um SELECT diretamente dentro de uma instrução INSERT. A sintaxe dessa operação é:

```
INSERT INTO banco.tabela-destino (c1, c2, ...)
  SELECT c1, c2,...
  FROM banco.tabela-origem;
```

A determinação dos campos tanto no INSERT quanto no SELECT são opcionais e deve ser usada quando apenas parte dos campos for copiada.

A figura 7.13 traz um exemplo onde uma tabela é gerada para armazenar o número de empregados e a soma dos salários pagos somente dos departamentos com mais de dois empregados. Esse tipo de comando pode ser muito útil na produção de relatórios Adhoc e em aplicações de Business Inteligence e Tomada de Decisão.

```
INSERT INTO deptoinfo
  SELECT d.dnumero, d.dnome, COUNT(e.ssn),
SUM(e.salario)
  FROM departamento d
  INNER JOIN empregado e ON e.dno = d.dnumero
  GROUP BY d.dnumero
  HAVING COUNT(e.ssn) > 2;
```

dnumero character	dnome character varyin	count bigint	sum double p
4	Administracao	3	93000
5	Pesquisa	4	133000

Figura 7.13
Inclusão de dados com uso de SELECT.

É possível também fazer a inserção de registros durante a criação de uma nova tabela no banco de dados. Essa funcionalidade geralmente é utilizada para realização de uma cópia dos dados antes de se realizar algum processamento na tabela de origem. Assim, pode-se criar uma tabela de testes que poderá ser descartada após o uso, sem a preocupação de manipulação indevida dos dados da tabela original. Vejamos um exemplo:

```
CREATE TABLE bkpEmpregado AS
  (SELECT * FROM empregado);
```

Outra possibilidade é fazer a atualização de dados em uma tabela tendo como base o resultado de um SELECT executado com o objetivo de filtrar os registros que serão atualizados. No exemplo a seguir, temos uma situação onde se deseja atualizar os valores dos produtos do tipo higiene com incremento de 4%.

```
UPDATE Produto
SET valor = valor * 1.04
WHERE tipocod IN (SELECT codtipo
                  FROM Tipo
                  WHERE nometipo = 'higiene');
```

Atividade de Prática 

Índices

- Servidor mantém uma **lista** de registros para cada tabela (não é ordenado)
- Na busca utiliza ***table scan*** (varredura na lista)
  - Funciona bem para poucos registros. Se houver poucos milhões de registros, pode não responder em tempo razoável
- Pode ser necessário ajuda de **índices**: 'tabelas especiais' mantidas em ordem específica
  - Não contém todos os dados, somente a(s) coluna(s)
  - Informações (**ponteiros**) sobre a localização das linhas fisicamente

Índices são usados para melhorar o desempenho dos bancos de dados. Um índice permite ao servidor de banco de dados encontrar linhas ou tuplas de uma tabela específica com muito mais rapidez do que faria sem o índice.

Esse benefício, porém, acaba por produzir um trabalho adicional para o SGBD, pelo fato de ter de se manter estruturas que indicam quais são os índices existentes e que poderão ser manipulados pelas aplicações que acessam as tabelas correspondentes. De qualquer modo, os índices podem melhorar o desempenho de operações envolvendo junções, atualizações e exclusões que contenham condição de procura.

O PostgreSQL implementa quatro tipos de índices, cada um com o seu grau de eficiência, e que devem ser utilizados de acordo com as características da aplicação que deles fará uso. São eles:

- **B-tree**: é o tipo padrão (assume quando não indicamos). São índices que podem tratar consultas de igualdade e de faixa, em dados que podem ser classificados. Indicado para consultas com os operadores: <, <=, =, >=, >. Também pode ser utilizado com LIKE, ILIKE, ~ e ~*;
- **R-tree:** tipo mais adequado a dados espaciais. Adequado para consultas com os operadores: <<, &<, &>, >>, @, ~=, &&;
- **HASH**: indicados para consultas com comparações de igualdade simples. Atualmente recomenda-se o B-tree em seu lugar;
- **GiST**: utilizado para retornar resultados com perdas, isto é, retorna os dados mais rapidamente comparando por coincidências de bits, prefira usar b-tree que é mais confiável.

Os tipos B-tree e GiST suportam índices com várias colunas. Índices com mais de um campo somente serão utilizados se as cláusulas com os campos indexados forem ligadas por AND. Um índice com mais de três campos dificilmente será utilizado.

Os arquivos que contêm as informações dos registros de uma determinada tabela são gravados nos servidores como uma lista. Inicialmente essa lista pode ser consultada por meio de uma varredura conhecida como table scan, tendo bom funcionamento com uma quantidade reduzida de registros (a consulta percorre cada um dos registros na tabela).

Com o crescimento do conjunto de informações gravadas em uma tabela, o recurso de table scan acaba se tornando ineficiente, ponto a partir do qual devemos começar a pensar no uso de índices.

Índices são estruturas que foram desenvolvidas com o objetivo único de facilitar e agilizar o processo de manutenção de informações armazenadas nas tabelas de um banco de dados.

- Por que então não indexar todas as tabelas em um SGBD?
- Por que não indexar tudo?
  - ESPAÇO: Todo índice é um tipo especial de tabela
  - PROCESSAMENTO: Qualquer inclusão/remoção na tabela (e até atualização que afete a(s) coluna(s) do índice), altera também o índice (atualização de ponteiros)
- Portanto, em casos especiais: indexar, executar a rotina e remover o índice
- Manter índices para chaves (primária/estrangeira)
- Indexar quaisquer colunas que serão frequentemente empregadas em consultas
  - coluna do tipo data é boa candidata

Chamamos a atenção para o fato de que há um limite para uso de índices. Um índice, antes de mais nada, é um tipo especial de tabela que utilizará espaço físico em disco para armazenar suas informações. Outro ponto importante é que os índices sempre serão consultados e atualizados durante o processamento das informações que estão em suas tabelas.

Dessa forma, orientamos para o uso dos campos que representam chaves primárias ou estrangeiras, ou mesmo algumas colunas que serão frequentemente empregadas em consultas, como possíveis candidatos a índices de tabelas no banco de dados.

Na realidade, o ideal é avaliar a necessidade de criação de cada novo índice. Para tanto, podemos medir o tempo para realizar algumas consultas sem o uso do índice e depois comparar se existem ganhos na realização das mesmas consultas fazendo uso do índice. Se os ganhos não forem significativos, considere excluir o índice criado.

Os SGBDs mais modernos possuem um otimizador de consultas que ajuda a determinar a maneira mais eficiente de executar operações. No PostgreSQL podemos utilizar o comando *EXPLAIN*, que permite visualizar todas as etapas envolvidas no processamento de uma consulta SQL/DML. Essa visão do funcionamento interno do banco permite fazer melhorias em consultas que estejam tomando tempo excessivo.

A sintaxe do EXPLAIN é:

```
EXPLAIN [ANALYZE] [VERBOSE] <consulta SQL>

Onde:

ANALYSE executa de fato o comando e retorna o tempo de execução.

VERBOSE vai detalhar ainda mais o resultado do comando EXPLAIN.
```

Na figura 7.14, temos um exemplo do uso do comando *EXPLAIN* e o resultado exibido após sua execução.

```
EXPLAIN
   SELECT pnome, unome
   FROM empregado e
   INNER JOIN dependente d ON e.ssn = d.essn
          WHERE e.sexo = d.sexodep;
```

QUERY PLAN text
Hash Join (cost=1.20..17.16 rows=1 width=156)
Hash Cond: ((d.essn = e.ssn) AND (d.sexodep = e.sexo))
-> Seq Scan on dependente d (cost=0.00..13.40 rows=340 width=48)
-> Hash (cost=1.08..1.08 rows=8 width=204)
-> Seq Scan on empregado e (cost=0.00..1.08 rows=8 width=204)

Figura 7.14
Resultado do EXPLAIN.

Já a figura 7.15 demonstra o uso do EXPLAIN com a cláusula ANALYZE, fazendo com que o resultado obtido traga também os tempos envolvidos em cada etapa.

```
EXPLAIN ANALYZE
   SELECT pnome, unome
   FROM empregado e
   WHERE e.ssn IN (SELECT d.essn
                   FROM dependente d
                   WHERE e.sexo = d.sexodep);
```

```
QUERY PLAN
text
Seq Scan on empregado e  (cost=0.00..1366.60 rows=95 width=156) (actual time=0.128..0.231 rows=2 loops=1)
  Filter: (SubPlan 1)
  Rows Removed by Filter: 6
  SubPlan 1
    ->  Seq Scan on dependente d  (cost=0.00..14.25 rows=2 width=40) (actual time=0.006..0.009 rows=3 loops=8)
          Filter: (e.sexo = sexodep)
          Rows Removed by Filter: 3
Total runtime: 0.337 ms
```

Figura 7.15
Resultado do EXPLAIN ANALYZE.

Criando, renomeando e removendo índices

Podemos utilizar a sintaxe a seguir para a criação explicitas de índices no SGBD, lembrando que no PostgreSQL índices são automaticamente criados quando se define uma PRIMARY KEY ou FOREIGN KEY durante a criação da estrutura da uma nova tabela.

```
CREATE [UNIQUE] INDEX <nomeIDX> ON <nomeTB>
[USING Algoritmo] (nomeCOL1 [ASC|DESC]
[NULLS {FIRST|LAST}] [,nomeCOL2...]);
```

Onde:

- **UNIQUE**: não permite a ocorrência de dados duplicados (índice exclusivo);
- **nomeIDX**: nome da estrutura de índice que será criada;
- **nomeTB**: nome da tabela que contém a coluna na qual será criada o índice de acesso;
- **Algoritmo**: método utilizado para manter o índice e utilizado na busca (BTREE, GIN, GIST e HASH), sendo BTREE o método escolhido por padrão. A escolha do algoritmo dependerá do total de registros existentes, variabilidade entre valores da coluna, entre outros;
- **nomeCOL**: nome da(s) coluna(s) sobre a(s) qual(is) será o criado índice (pode ser de parte do campo, com uso de funções);
- **Opção ASC/DESC**: indica se a criação do índice será ordenada de forma crescente (ASC)-padrão ou decrescente (DESC);
- **NULLS {FIRST|LAST}**: especifica que nulos serão ordenados primeiro ou no final. Quando DESC é utilizado, nulos são os primeiros por padrão.

É possível também criar um índice através do assistente do PostgreSQL, conforme demonstrado na figura 7.16.

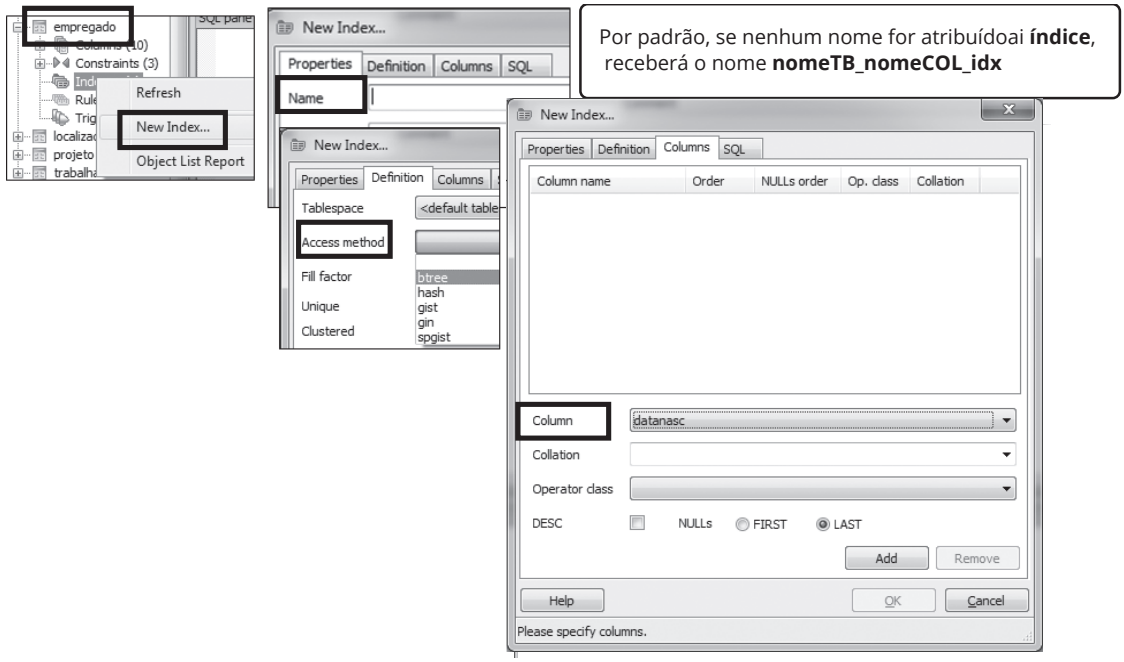


Figura 7.16
Criação de índice pelo assistente.

A sintaxe dos comandos para renomear (*ALTER.. RENAME*) e excluir (*DROP*) um índice é bastante simples, conforme mostramos a seguir:

```
ALTER INDEX <nomeIDX> RENAME TO <novoNomeIDX>;

DROP INDEX [ IF EXISTS ] <nomeIDX> [CASCADE|RESTRICT];
```

No *DROP*, a cláusula IF EXISTS restringe a emissão de mensagens de erro caso os índices a serem excluídos não existam. O padrão é a emissão de erro ao se tentar excluir um índice inexistente. As demais cláusulas, que são opcionais, explicadas a seguir:

- **CASCADE**: remove objetos que dependam do índice;
- **RESTRICT**: faz com que o índice não seja eliminado se existirem objetos filhos, sendo esta a opção padrão do PostgreSQL.

Visões

Visão (view): mecanismo de consulta de dados

- Qualquer relação/tabela que não faça parte do modelo conceitual mas seja visível a um usuário como uma relação/tabela virtual
- Armazena a estrutura do script SELECT da SQL
  - É um componente do BD (tabela virtual) como sequência, índice, etc.
- É uma tabela 'lógica', composta de linhas de uma ou mais tabelas (agrupadas ou não) e não ocupa espaço no BD
- Produz uma tabela temporária (resultSet) ou cursor, a partir das tuplas das tabelas básicas (que existem de forma física no BD)

Em SQL, uma visão (view) é uma tabela única derivada de outra(s) tabela(s), não existindo fisicamente e por isso mesmo sendo considerada uma tabela virtual.

Visões são comumente utilizadas na criação de tabelas temporárias que precisam ser consultadas frequentemente, sem que haja a necessidade de ter uma representação física.

Aplicações bem projetadas expõem uma interface pública que mantém privados os detalhes de sua implementação. Dessa forma, mudanças futuras no projeto não impactam o usuário final. Portanto, uma visão também pode ser vista como um mecanismo de consulta de dados, tendo como resultado uma tabela virtual com relações ou tabelas que não fazem parte de modelo conceitual.

As principais aplicações para visões são:

- **Segurança de dados**: exibir partes da tabela;
- **Agregação de dados**: relatório comum já "montado" pelo projetista do BD sem intervenção do desenvolvedor;
  - Exemplo: mensalmente mostrar a conta e o total de depósitos.
- **Esconder complexidade**: só executa a subconsulta se a coluna for informada;
- **Juntar dados particionados**: tabelas separadas com dados atuais e de histórico (UNION).

Criação de visão

Em SQL, o comando para especificar ou criar uma visão é o *CREATE VIEW*. A visão deverá receber um nome, uma lista de atributos e uma instrução SQL, geralmente uma consulta SELECT, para especificar o conteúdo dessa visão.

```
CREATE [OR REPLACE] VIEW <nomeVW> [(nomeCOL1, ...)] AS <query> ;
```

Onde:

- **nomeVW**: nome da visão a ser criada;
- **nomeCOL**: relação opcional de nomes a serem usados para as colunas da visão;
  - Quando fornecidos, esses nomes substituem os nomes inferidos a partir da consulta SQL.
- **query**: uma consulta SQL (ou seja, um comando *SELECT*) que gera as colunas e as linhas da visão.

Uma visão pode ser também criada com o auxílio do assistente do pgAdmin3, conforme demonstrado na figura 7.17.

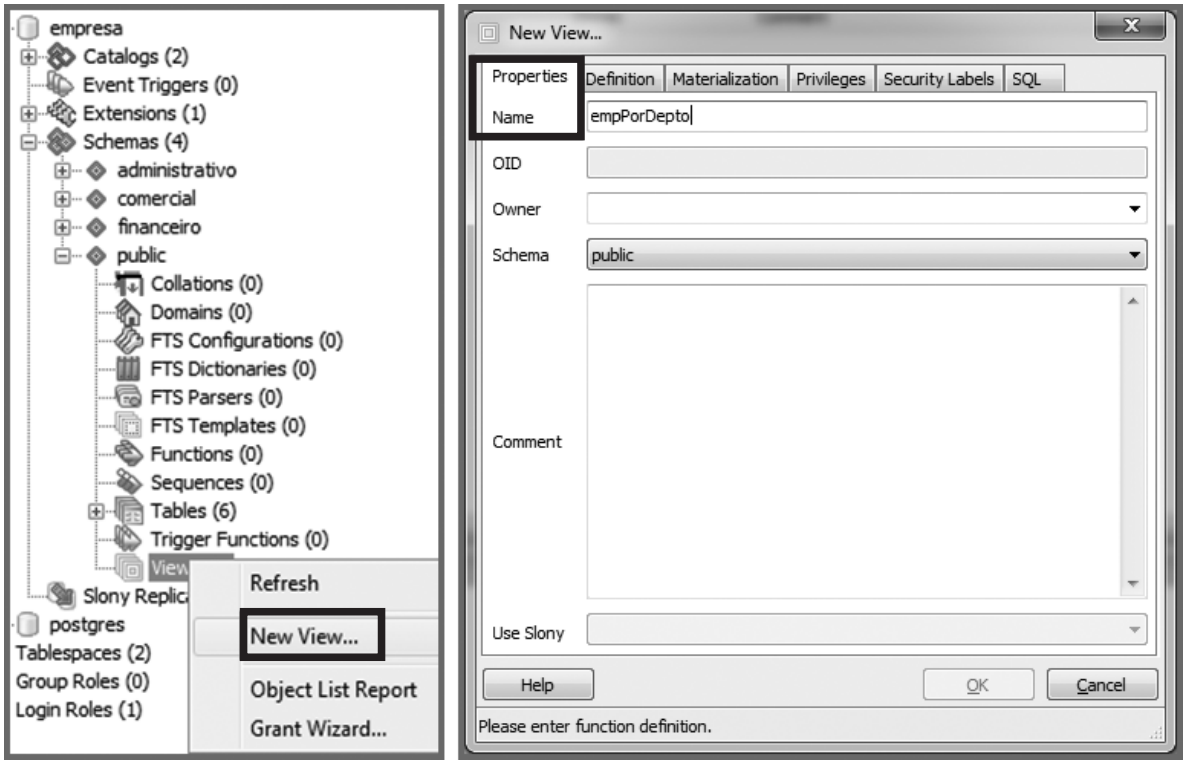


Figura 7.17
PgAdmin III – Assistente para criação de visões.

No exemplo a seguir, cria-se uma visão com os nomes dos departamentos e seus respectivos empregados.

```
CREATE OR REPLACE VIEW "empPorDepto" AS
  SELECT d.dnome AS departamento,
   e.pnome || ' ' || e.inicialm || ' ' ||
   e.unome AS empregado
  FROM departamento d
  JOIN empregado e ON e.dno = d.dnumero
  ORDER BY 1, 2;
```

Existem algumas restrições na criação de visões, similares às restrições existentes para a criação de tabelas, tais como:

- Não pode haver no resultado duas colunas com o mesmo nome;
- Não pode referenciar variáveis de sistema.

Outro ponto importante a se destacar é que não é possível a inserção de valores dentro de uma visão. Novos valores só aparecerão dentro da visão quando as tabelas que a compõem tiverem novos registros ou mesmo uma atualização sobre os registros já existentes.

Executando uma visão

Podemos especificar uma consulta SQL em uma visão da mesma forma que especificamos as consultas envolvendo as tabelas de um banco de dados. Dessa forma, podemos selecionar total ou parcialmente os dados retornados pela visão, ou mesmo combinar a visão com qualquer outra cláusula e junto com outras visões e tabelas. A figura 7.18 mostra um comando *SELECT* sendo aplicado em uma visão através do assistente do pgAdmin3.

Executando uma visão

```
SELECT * FROM <nomeVW);
```

▣ Combinando com qualquer outra cláusula e junto com outras visões e tabelas

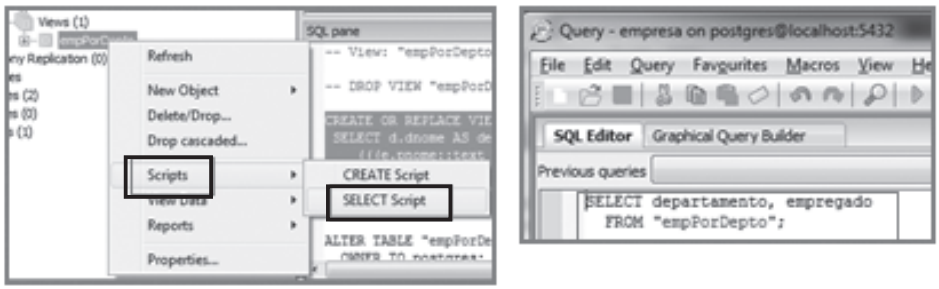
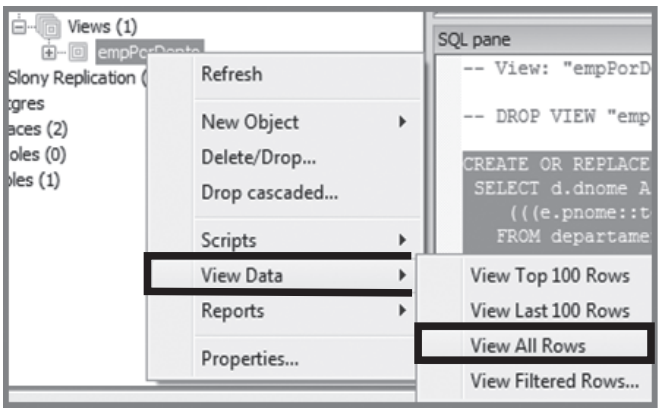


Figura 7.18
PgAdmin III – Consulta em uma visão.

A figura 7.19, por sua vez, mostra como consultar as informações contidas em uma visão através do pgAdmin3.

```
SELECT * FROM empPorDepto;
```



departamento character varying(30)	empregado text
Administracao	Alberto V Medeiros
Administracao	Alice J Zebra
Administracao	Jennifer S Wallace
Pesquisa	Fabio T Will
Pesquisa	Joao B Souza
Pesquisa	Jussara A Pereira
Pesquisa	Ricardo K Nantes
Sede Administrativa	Joaquim E Brito

Figura 7.19
Resultado da execução da visão empPorDepto.

Alterando e removendo uma visão

É possível alterar uma visão por meio do comando de criação, adicionando o parâmetro de substituição REPLACE, como segue:

```
CREATE OR REPLACE VIEW <nomeVW> ... ;
```

O parâmetro REPLACE indicará para o SGBD que a view deverá ser atualizada, sendo possível atualizar seu nome ou a composição de sua consulta SQL.

Já a exclusão da visão pode ser feita pelo comando a seguir:

```
DROP VIEW <nomeVW>;
```

Excluir uma visão não exclui os dados das tabelas aos quais os mesmos pertencem. Somente a estrutura da visão será descartada.

8

Introdução à programação SQL (pl/pgsql)

objetivos

Obter visão geral da programação usando a linguagem PL/pgSQL e possibilidades de sua aplicação no desenvolvimento de procedimentos armazenados e na manipulação de dados.

conceitos

Programação; Procedimentos armazenados; Declaração de parâmetros; Estruturas de controle; PL/pgSQL.

Exercício de nivelamento

Você já trabalhou com alguma linguagem que possibilitou a manipulação de dados via procedimentos armazenados? Qual linguagem utilizou? Em qual SGBD?

PL/pgSQL

A linguagem estrutural PL/pgSQL, desenvolvida como uma extensão da linguagem SQL, tem como principal objetivo permitir a programação de tarefas no PostgreSQL. A incorporação de características procedurais à SQL agregam benefícios e facilidades de controle de fluxo de execução de instruções, tais como loops estruturados (for, while) e controle de decisão (if then else).

Os objetivos de projeto da linguagem PL/pgSQL foram no sentido de criar uma linguagem procedural que pudesse:

- Ser utilizada para criar procedimentos de funções e de gatilhos;
- Adicionar estruturas de controle à linguagem SQL;
- Realizar processamentos complexos;
- Herdar todos os tipos de dados, funções e operadores definidos pelo usuário;
- Ser definida como confiável pelo servidor;
- Ser fácil de utilizar.

Dessa forma, programar em PL/pgSQL significa ter à disposição um ambiente procedural totalmente desenvolvido para aplicações de bancos de dados, beneficiando-se do controle transacional inerente aos SGBDs.

Outro ponto a se considerar é que PL/pgSQL não é a única linguagem que podemos usar dentro do PostgreSQL. Além dela, podemos citar PL/Tcl, PL/PERL, PL/Python, entre outras. Essas linguagens podem ser instaladas como módulos adicionais ao PostgreSQL (caso não estejam presentes na sua instalação padrão). De modo geral, todas essas linguagens permitem o uso de DMLs (CRUD) e instruções de controle de transação (COMMIT, ROLLBACK e SAVEPOINT).

Características do PL/pgSQL:

- Capacidade procedural (comandos de controle e repetição)
- Redução de tráfego de rede
- Permite DMLs (SELECT, INSERT, UPDATE, DELETE)
- Instruções de controle de transação (COMMIT, ROLLBACK, SAVEPOINT)
- No PostgreSQL 9.x o PL/pgSQL é instalado automaticamente
  - É possível remove-lo (módulo carregável), mas não é recomendável.

Linguagens procedurais no PostgreSQL:

- PL/pgSQL, PL/Tcl, PL/PERL, PL/Python

No PostgreSQL 9.0, o PL/pgSQL é instalado automaticamente, embora seja possível removê-lo (é visto como um módulo carregável). Alguns administradores podem querer excluir essa funcionalidade do SGBD, mas entendemos que isso não é recomendável.

Funções

- Funções são blocos de código SQL armazenados no servidor do banco de dados
- Podem ser invocados a qualquer momento com o objetivo de realizar algum processamento
- Procedimentos encarregam-se da lógica de negócio e não retornam valor
  - No PostgreSQL funções estão mescladas com os procedimentos

Funções devem ser vistas como blocos de código contendo instruções SQL que estão localizados do lado do servidor. Um dos objetivos alcançados com o uso desse recurso é o de diminuição de tráfego de rede, uma vez que determinados controles e fluxo de execução podem ser transferidos para dentro do SGBD, não sendo necessária sua implementação em interfaces de acesso ao banco de dados.

Uma vez definidas as funções, podem ser invocadas a qualquer tempo, sempre que for necessária a realização do processamento de suas instruções SQL no banco de dados.

Na especificação do PosgreSQL, funções estão mescladas com os procedimentos, sendo os procedimentos encarregados da lógica de negócios sobre as tabelas de um terminado banco de dados, não retornando valores após sua chamada. Já as funções podem ou não retornar valores após a realização de suas lógicas de negócios.

Criação funções

A seguir, apresentamos a estrutura sintática utilizada em PL/pgSQL para a criação de funções.

```
CREATE [OR REPLACE] FUNCTION <nomeF>( [<argumentos>] )

RETURNS <tipoRetorno> AS $$ [<<label>>]
```

```
[DECLARE
   /* declaração de variáveis, constantes, ... */]
BEGIN
   /* Seção das instruções SQL */
   [RETURN <valor>;]
END [label];
$$
LANGUAGE <linguagem>;
```

Onde:

- OR REPLACE: atualiza uma função se esta já existe;
- $$ são conhecidos como delimitadores de string que indica o início e fim da função SQL;
- <<label>> e label são delimitadores de blocos de uso não obrigatório;
- LANGUAGE: especifica o tipo da linguagem utilizada na escrita da função.

Blocos são usados para agrupamento lógico de instruções SQL que fazem parte da função sendo desenvolvida, não sendo obrigatória o seu uso.

Quantos aos comentários, podemos incluí-los dentro do processo de escrita das funções, sendo que estes seguem as mesmas regras da linguagem SQL.

Por padrão, todas as palavras-chave utilizadas no processo de estruturação de uma função são case-insensitive, ou seja, tanto faz escrevê-las com letras maiúsculas ou minúsculas.

A sintaxe de comentários é apresentada a seguir:

```
-- para comentar o restante da linha
/* [bloco] */ para escrever um comento em bloco, com mais de uma linha
```

A figura 8.1 traz um exemplo de função cujo retorno é vazio (VOID). Nesse exemplo, $BODY$ é o delimitador de string indicando o início e fim dos comandos que compõem a função. Temos ainda o comando de saída RAISE NOTICE, utilizado para enviar uma "mensagem" para o console padrão, onde % funciona como um caractere coringa a ser substituído por um valor que vai compor a mensagem.

```
CREATE OR REPLACE FUNCTION teste()
RETURNS void AS
$BODY$
DECLARE din REAL; taxa REAL;
BEGIN
  din := 100; taxa := 0.15;
  RAISE NOTICE 'Rendimento de R$ %', (din * taxa);
END;
$BODY$
LANGUAGE plpgsql;
```

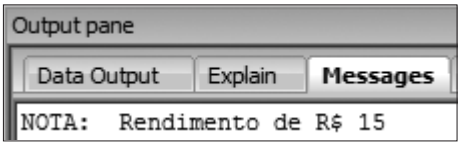


Figura 8.1
RAISE NOTICE: exemplo de mensagem via console.

Declaração, inicialização e atribuição

Uma funcionalidade importante dentro do processo de construção de funções é a possibilidade de utilização de identificadores, que compreende as etapas de declaração, inicialização e atribuição de valores iniciais. Identificadores são vistos como variáveis ou constantes, entre outros recursos (disponibilizados pelo PostgreSQL) que podemos usar durante o processamento das instruções de uma determinada função.

A sintaxe da instrução para inicialização de uma variável é:

```
<nomeId> [CONSTANT] <tipo> [NOT NULL]
    [{DEFAULT|:=} <expressão>];
```

Onde <nomeId> é o nome utilizado para identificar a variável, seguido da cláusula CONSTANT (que é opcional) quando o valor atribuído à variável for fixo. Em seguida temos <tipo>, indicando o tipo da variável como, por exemplo, numérico, real, ou até mesmo um tipo específico de uma estrutura criada. Já a cláusula DEFAULT, também opcional, indicará o valor inicialmente assumido pela variável. O operador := funciona da mesma forma, indicando a atribuição de um valor ou resultado de uma expressão para a variável em questão.

No exemplo a seguir, temos a declaração de algumas variáveis que serão utilizadas pela função Teste2.

```
CREATE OR REPLACE FUNCTION teste2()
RETURNS void AS
$BODY$
   DECLARE  vNome VARCHAR;
   raio NUMERIC := 5;
   pi CONSTANT REAL := 3.1415927;
BEGIN
   RAISE NOTICE 'A área do círculo de raio % é %', raio, (pi * POW(
raio, 2));
   SELECT pjnome INTO vNome
   FROM projeto WHERE pnumero = '20';
   RAISE NOTICE 'Projeto 20: %', vNome;
END;
$BODY$
LANGUAGE plpgsql VOLATILE;
```

Ainda sobre o exemplo anterior, podemos ver o uso de POW(), que é uma função nativa do PostgreSQL. Ou seja, não há nenhum problema em usar funções no processo de criação de novas funções. Além disso, percebam que a cláusula INTO é utilizada junto com um SELECT, permitindo que o resultado obtido inicialize a variável vNome.

Por fim, na última linha do exemplo, aparece a cláusula VOLATILE.

Na verdade, o PostgreSQL disponibiliza três possíveis "categorias" para as funções, indicando como será o comportamento em relação ao banco de dados. São elas:

- **VOLATILE (padrão)**: que pode alterar os dados do banco;
- **STABLE**: não modifica a base de dados e retornará os mesmos valores caso sejam fornecidos os mesmos parâmetros na mesma chamada;
- **IMMUTABLE**: não altera jamais a base de dados e sempre retornará os mesmos valores para os mesmos parâmetros.

Utilize VOLATILE se a função for usar *INSERT*, *UPDATE* ou *DELETE* (altera os dados da base). O uso de STABLE é indicado em instruções SELECT internas, onde seu resultado pode variar a cada chamada da função. Já IMMUTABLE é usado nos demais casos, como funções sem uso de comando *SELECT*, que realizam cálculos matemáticos, entre outras operações que não alteram a base de dados.

A sintaxe para executar uma função qualquer é bastante simples:

```
SELECT <nome_da_funçao>();
```

Basta utilizar o comando *SELECT* seguido do nome da função. Se esta fizer uso de parâmetros, esses deverão ser referenciados entre parênteses, logo após o nome da função.

Não custa lembrar, contudo, que também podemos criar e executar funções utilizando o assistente do pgAdminIII. Na figura 8.2, podemos ver a janela exibida quando acionamos o menu flutuante (com o botão esquerdo do mouse) para criar uma nova função. Essa janela trás uma série de abas onde são informadas as propriedades e características da função.

As instruções que vão compor a função devem ser informadas na aba CODE.

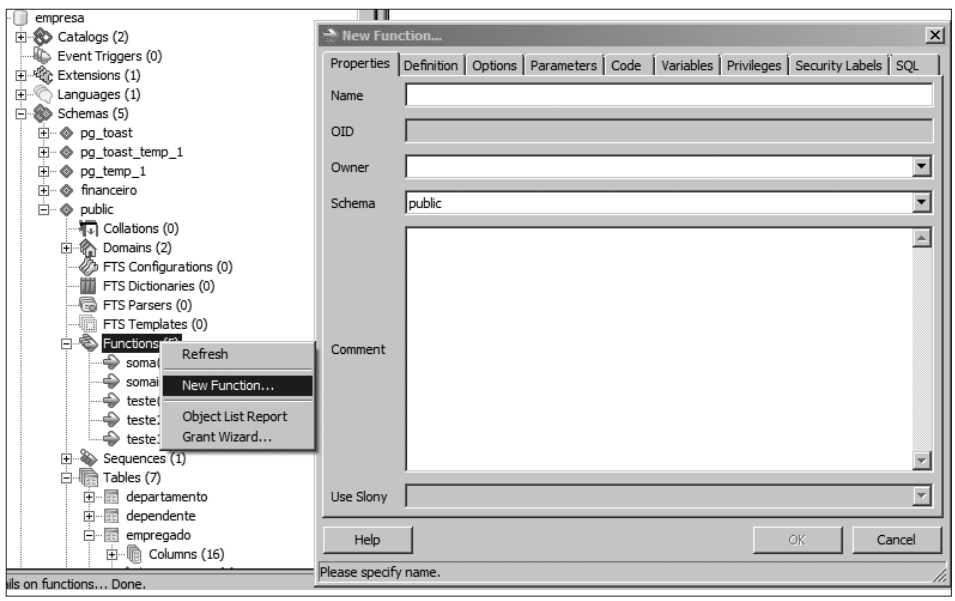


Figura 8.2
pgAdminIII: assistente para criar de funções.

A figura 8.3, por sua vez, mostra os passos que devem ser observados para executar uma função pelo assistente.

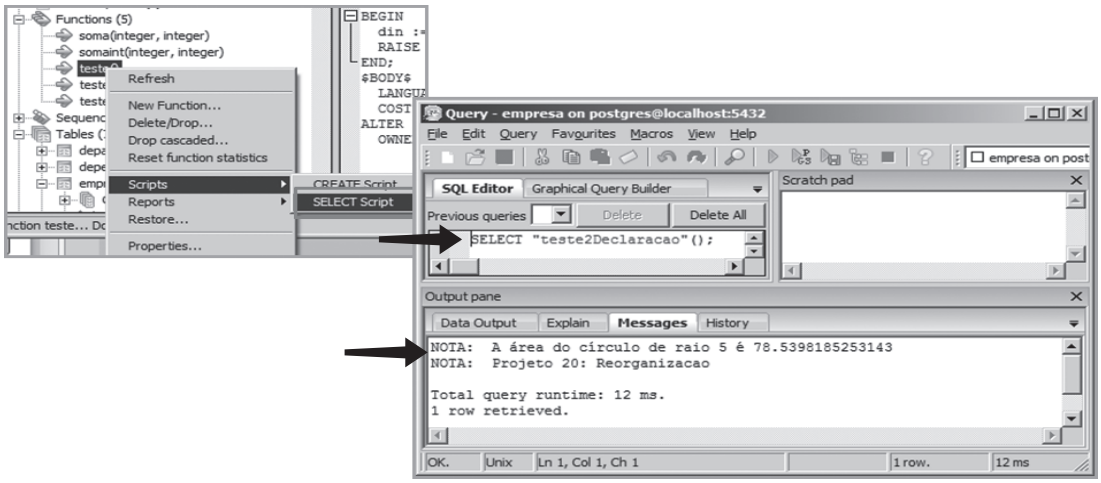


Figura 8.3
pgAdminIII: assistente para execução de funções.

Outro aspecto a ser lembrado no processo de declaração de variáveis diz respeito à possibilidade de especificarmos o tipo de dado de uma variável tomando como base o tipo de um campo existente em uma tabela no banco de dados. Ao fazer isso, temos uma preocupação a menos, já que passa a se responsabilidade do SGBD identificar e utilizar o tipo de dado correto quando da execução da função.

Mostramos a seguir a sintaxe para esse tipo de declaração, seguido de um exemplo de seu uso:

Declarar uma variável do mesmo tipo de dado de uma coluna da tabela

- Utilização da cláusula **%TYPE**

```
<nome_variavel> <tabela>.<campo>%TYPE ;
```

- exemplo:

```
DECLARE
  vNome PROJETO.pjnome%TYPE;
  v REAL := cell(2.15) ;
BEGIN
  RAISE NOTICE 'x = %', x ;
  SELECT pjnome INTO vNOME FROM projeto
   WHERE pnumero '20';
  RAISE NOTICE 'Projeto 20: %', vNome;
END;
```

Basicamente, após o nome da variável, é preciso indicar respectivamente o nome da tabela e campo, seguidos da cláusula %TYPE.

Declaração de parâmetros

Ao declarar parâmetros considere:

- Parâmetros passados para as funções são declarados por apelidos.
- Podem ser referenciados pelo apelido ou $1, $2, e assim por diante, na ordem informada nos parênteses.

Uma função pode fazer uso de parâmetros, responsáveis por receber possíveis valores durante a sua chamada, e que serão utilizados para realização de algum processamento interno.

O exemplo a seguir ilustra a declaração e uso do parâmetro taxa:

```
CREATE OR REPLACE FUNCTION teste4(taxa REAL) RETURNS void AS
$BODY$  DECLARE valor REAL := 100 ;
BEGIN
  RAISE NOTICE 'A aplicação de R$100 com taxa de % rendeu R$ %',
    taxa, (valor * taxa);
END; $BODY$ LANGUAGE plpgsql;
```

Observe que o comando:

```
    taxa, (valor * taxa);
```

Pode ser substituído por:

```
    taxa, (valor * $1);
```

Dentro do corpo da função, podemos referenciar o parâmetro pelo seu nome ou pelo identificador $ seguido da posição que esse ocupa, da esquerda para a direita, na sequência de parâmetros declarados entre parênteses após o nome da função. Reforçamos esse conceito com mais um exemplo:

```
CREATE OR REPLACE FUNCTION teste5soma(IN x real, IN y real, OUT r
real) RETURNS real AS
$BODY$
BEGIN
  -- r := x + y ;
  r := $1 + $2 ;
  RAISE NOTICE 'Resultado %', r;
END;
$BODY$
LANGUAGE plpgsql;
```

Destacamos, no exemplo anterior, o uso de IN e OUT na declaração dos parâmetros. Essas cláusulas indicarão o modo como cada parâmetro será utilizado, admitindo as seguintes possibilidades:

Modos do parâmetro:

- IN: entrada;
- OUT: saída;
  - Variável que começa NULL;
  - Deve ser atribuído um valor durante a execução da função;
  - O valor final do parâmetro é retornado.
- IN OUT: entrada e saída;

- VARIADIC: número variado de argumentos;
  - Array.
  - Devem ser do mesmo tipo de dado.

A figura 8.4 a seguir exibe a aba “Parameters” da janela onde são declarados os parâmetros de uma função através do assistente

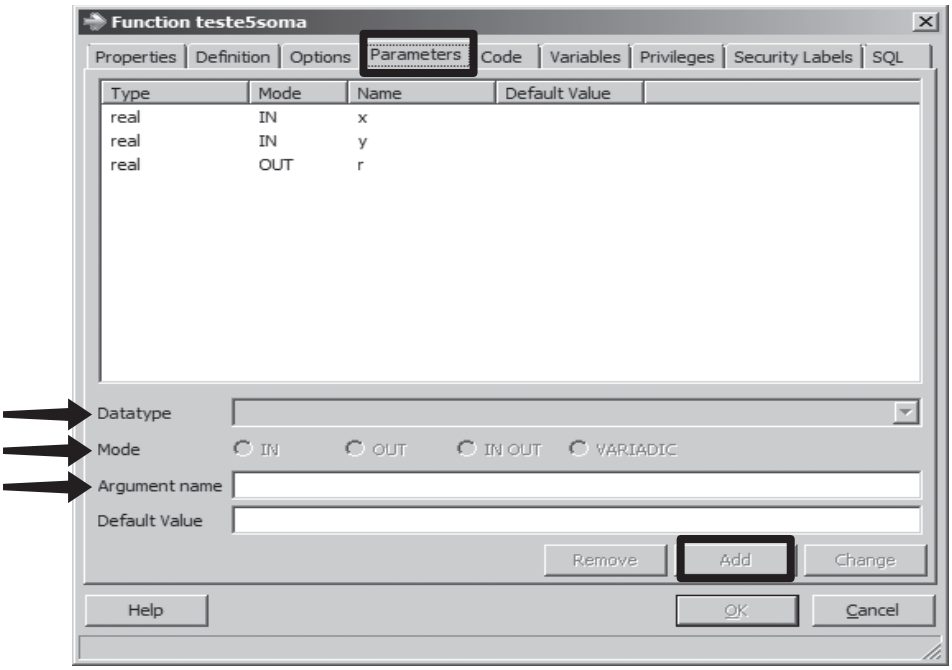


Figura 8.4
pgAdminIII: assistente para declaração de parâmetros.

Já a figura 8.5 trás janelas com o resultado da execução da função teste5soma(), apresentada em um exemplo anterior. Atente para o fato de que não aparece no corpo da função a palavra reservada RETURN, já que o valor a ser retornado fica definido pelo uso de OUT na declaração da variável r.

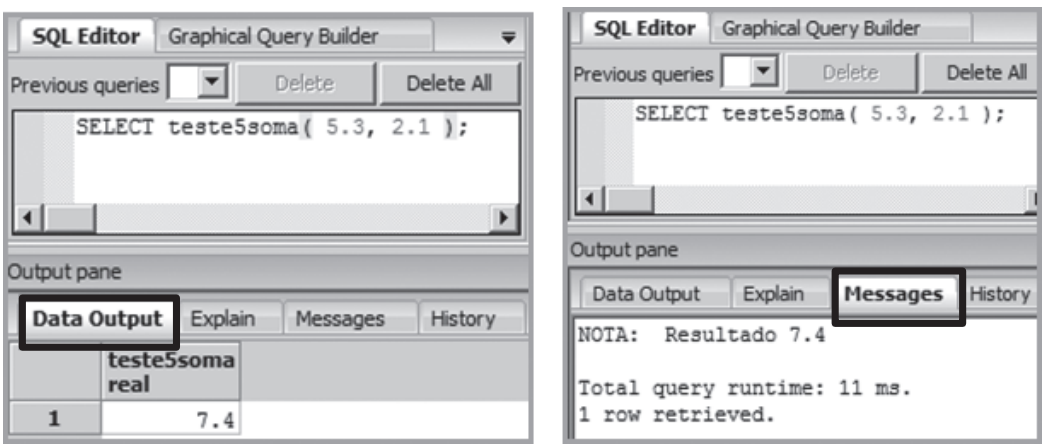


Figura 8.5
Data Output e Messages.

Variável composta heterogênea (ou tipo-linha)

- Variável que pode armazenar toda uma linha de um SELECT (resultado da consulta)
- Os campos individuais do valor linha são acessados utilizando a notação de ponto <variável_linha>.campo
- É possível a utilização da cláusula %ROWTYPE para declarar uma variável do mesmo tipo de dado de uma coluna da tabela

Assim como na linguagem de programação, funções em PL/pgSQL também permitem a manipulação de variáveis compostas heterogêneas, ou tipo-linha, como são conhecidas no PostgreSQL.

Esse tipo de variável pode armazenar toda uma linha resultante de uma instrução SELECT, desde que o conjunto de colunas do comando corresponda ao tipo declarado para a variável.

Após o armazenamento dos valores, podemos utilizar a notação de ponto para manipular individualmente os campos resultantes do SELECT.

```
<variável_linha>.campo
```

Também temos a possibilidade de usar a cláusula %ROWTYPE para declarar uma variável do mesmo tipo de dado de uma coluna da tabela.

```
<tabela>%ROWTYPE ;
```

A figura 8.6 trás um exemplo da declaração e uso de variáveis tipo-linha.

```
CREATE OR REPLACE FUNCTION teste6rowtype(dnum character)
  RETURNS text AS
$BODY$
DECLARE
  vDepto departamento%ROWTYPE;
BEGIN
  SELECT * INTO vDepto
    FROM departamento
     WHERE dnumero = dnum ;
  RETURN 'Nome Depto: ' || vDepto.dnome || ', Gerente: ' || vDepto.
gerssn;
END;
$BODY$
LANGUAGE plpgsql;
```

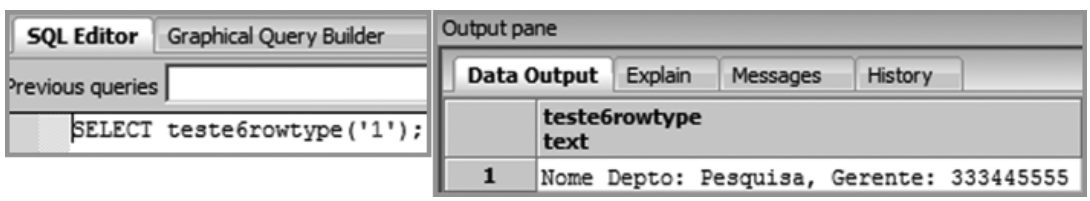


Figura 8.6 Console com mensagem utilizando %ROWTYPE.

Variável tipo RECORD (Registro)

- Variáveis de registro são semelhantes ao tipo-linha, mas elas não têm nenhuma estrutura pré-definida
- Assumem a estrutura da linha definida no SELECT
- A subestrutura de uma variávelregistro pode mudar cada vez que é atribuído
  - Consequência é que qualquer tentativa de acessar um campo não definido no momento irá apresentar um erro em tempo de execução`<identificador>`

Outro tipo disponível para declaração de variáveis dentro de funções em PL/pgSQL é o RECORD. Variáveis do tipo registro são semelhantes ao tipo-linha, sem a presença de uma estrutura pré-definida.

```
<identificador> RECORD ;
```

Isso faz com que a variável assuma a estrutura da linha definida na instrução SQL definida no corpo da função, podendo sofrer alteração a cada nova atribuição. Como consequência, qualquer tentativa de acessar um campo não definido no momento da atribuição dessa variável vai gerar um erro em tempo de execução.

Um exemplo com a declaração e uso de uma variável registro pode ser visto na figura 8.7.

```
CREATE OR REPLACE FUNCTION teste7record()
  RETURNS void AS
$BODY$ DECLARE
  vEmp RECORD ;
BEGIN
  SELECT (pnome||' '||inicialm||'. '|| unome) AS nome,
      TO_CHAR(datanasc, 'DD/MM/YYYY') AS data INTO vEmp
    FROM empregado
    WHERE ssn = '123456789';
  RAISE NOTICE 'Nome: % , DataNasc: %', vEmp.nome, vEmp.data;
END;$BODY$
LANGUAGE plpgsql;
```

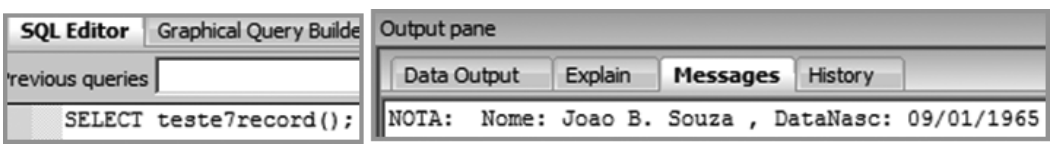


Figura 8.7
Console com mensagem utilizando RECORD.

Cláusula RETURNING

- Pode ser necessário recuperar valores em operações de atualização (INSERT/UPDATE/DELETE)
- Exemplo: recuperar o código do cliente (serial) inserido para que possa ser empregado em outra operação

Existem situações onde pode ser necessário recuperar determinados valores durante a execução de operações de atualização (INSERT/UPDATE/DELETE). Para tanto é utilizada a cláusula RETURNING, que no exemplo a seguir permite recuperar o código do cliente inserido para que possa ser empregado em outra operação.

```
CREATE OR REPLACE FUNCTION testeReturning(vcli VARCHAR)
  RETURNS void AS
$BODY$
   DECLARE vid INTEGER ;
BEGIN
  INSERT INTO cliente (codcli, nomecli)
    VALUES (DEFAULT, vcli) RETURNING codcli INTO vid;
  RAISE NOTICE 'ID = % ', vid ;
END;
$BODY$
LANGUAGE plpgsql;
```

A chamada para esta função será feita da seguinte forma:

```
SELECT “testeReturning”('SERGIO SILVA');
```

Produzindo o retorno:

```
ID = 7
```

Atividade de Prática AVA

Estruturas de controle

As estruturas de controle provavelmente são a parte mais útil (e mais importante) da linguagem PL/pgSQL. Essas estruturas possibilitam uma forma mais flexível e poderosa para a manipulação de dados no PostgreSQL. Apresentaremos a seguir algumas estruturas condicionais e de repetição.

Condicional IF

Permite executar comandos alternativos baseados em certas condições, de acordo com a seguinte sintaxe:

```
IF <expressãoCondicional> THEN
  <comandosSQL>;
  [ELSIF <expressãoCondicional2> THEN
    <comandosSQL2>;
      [ELSIF <expressãoCondicional3> THEN
        <comandosSQL3>; ...]]
          [ELSE <comandosSQLn>; ]
END IF;
```

Assim, podemos ter a execução de um conjunto de comandos baseado apenas em uma única condição, ou um conjunto aninhado de condições delimitando diversos blocos de comandos entre os quais serão executados apenas os que correspondem à condição que for avaliada como verdadeira.

```
IF ... THEN … END IF;
IF ... THEN ... ELSE … END IF;
IF ... THEN ... ELSIF ... THEN ... ELSE … END IF;
```

A figura 8.8 trás um primeiro exemplo onde a instrução IF.. THEN .. END IF é usada para determinar se o número passado como parâmetro é par ou é ímpar.

```
CREATE OR REPLACE FUNCTION teste9if(n integer)
  RETURNS character varying AS
$BODY$
BEGIN
  IF MOD(n,2) = 0
    THEN RETURN 'PAR';
  END IF;
  RETURN 'IMPAR';
END;
$BODY$
LANGUAGE plpgsql;
```

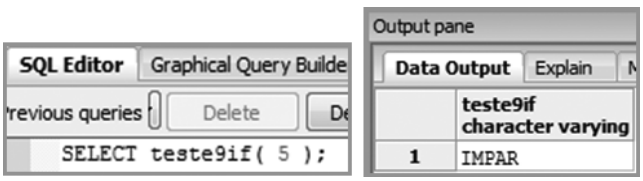


Figura 8.8
Resultado da execução da função teste9if().

Já o exemplo da figura 8.9 usa uma estrutura um pouco mais complexa para informar a situação de um determinado aluno de acordo com sua média.

```
CREATE OR REPLACE FUNCTION teste10elsif(media real)
  RETURNS void AS
$BODY$
DECLARE vSituacao VARCHAR;
BEGIN
  IF media <= 5
    THEN vSituacao := 'REGULAR';
    ELSIF media < 7
        THEN vSituacao := 'BOM';
   ELSE vSituacao := 'EXCELENTE';
  END IF;
  RAISE NOTICE 'Situação do aluno %', vSituacao;
END;
$BODY$
LANGUAGE plpgsql;
```

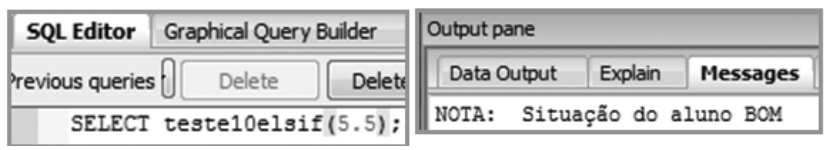


Figura 8.9
Resultado da execução da função teste10elsif().

Condicional CASE

A estrutura condicional CASE é outra forma para executar blocos de comandos com base em condições. Essa estrutura deve ser utilizada de acordo com a seguinte sintaxe:

```
CASE <expressão>
  WHEN <valor1> THEN <comandosSQL1>;
  [ WHEN <valor2> THEN <comandosSQL2>; ... ]
  [ ELSE <comandosSQLn>; ]
END CASE;
Ou
CASE
  WHEN <expressão> THEN <comandosSQL>;
  [ WHEN <expressão2> THEN <comandosSQL2>; ... ]
  [ ELSE <comandosSQLn>; ]
END CASE;
```

Um exemplo para tratar a opção de menu escolhida pode ser visto na figura 8.10.

```
CREATE OR REPLACE FUNCTION teste11case(opcao integer)
  RETURNS void AS $BODY$
DECLARE msg VARCHAR(20);
BEGIN
  CASE opcao
    WHEN 1 THEN msg := 'Opção de Saldo';
    WHEN 2 THEN msg := 'Opção de Extrato';
    WHEN 3 THEN msg := 'Opção Sair';
    ELSE msg := 'Opção inválida!!';
  END CASE;
  RAISE NOTICE 'Você escolheu %', msg;
END; $BODY$
LANGUAGE plpgsql;
```

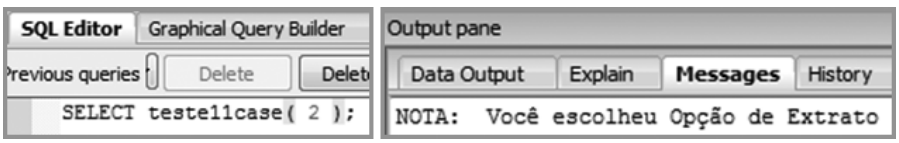


Figura 8.10
Resultado da execução da função teste11case().

Um segundo exemplo utilizando CASE, nesse caso para exibir a faixa do valor que é passado como parâmetro, pode ser visto na figura 8.11.

```
CREATE OR REPLACE FUNCTION teste12case(x real)
  RETURNS void AS $BODY$
DECLARE msg VARCHAR;
BEGIN
  CASE
    WHEN x BETWEEN 0 AND 10
      THEN msg := 'valor entre 0 e 10';
    WHEN x BETWEEN 11 AND 20
      THEN msg := 'valor entre 11 e 20';
    ELSE msg := 'valor menor que 0 ou maior que 20';
  END CASE;
  RAISE NOTICE '%', msg;
END; $BODY$
LANGUAGE plpgsql;
```

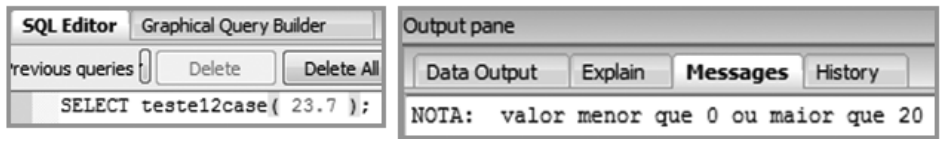


Figura 8.11
Resultado da execução da função teste12case().

Repetição

São estruturas que permitem repetir um conjunto de comandos até que se chegue a uma condição de saída (EXIT). A instrução LOOP obedece à seguinte sintaxe:

```
LOOP
  -- comandos
  IF <expressãoFIM>
    THEN EXIT;
  END IF;
END LOOP;
Ou
LOOP
  -- comandos
  EXIT WHEN <expressãoFIM>;
END LOOP;
```

Na figura 8.12, temos um exemplo onde o bloco de comandos é executado por dez vezes.

```
CREATE OR REPLACE FUNCTION teste13loop()
  RETURNS void AS $BODY$
DECLARE i INTEGER := 1;
BEGIN
  LOOP
    RAISE NOTICE '%', i;
    i := i + 1;
    EXIT WHEN i > 10;
  END LOOP;
END; $BODY$
LANGUAGE plpgsql;
```

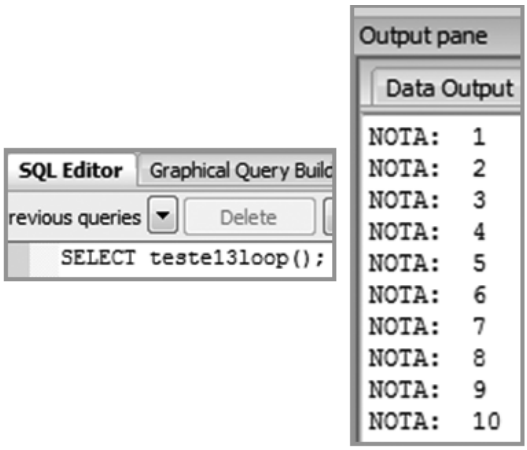


Figura 8.12
Resultado da execução da função teste13loop().

Outra estrutura de controle semelhante é o WHILE (enquanto), que tem a seguinte sintaxe:

```
WHILE <expressãoTRUE> LOOP
  -- comandos
END LOOP;
```

O exemplo da figura 8.13 apresenta o mesmo resultado do exemplo anterior, mas nesse caso utilizando o comando *WHILE*.

```
CREATE OR REPLACE FUNCTION teste14while()
  RETURNS void AS $BODY$
DECLARE i INTEGER := 1;
BEGIN
  WHILE i <= 10 LOOP
    RAISE NOTICE '%', i;
    i := i + 1 ;
  END LOOP;
END; $BODY$
LANGUAGE plpgsql;
```

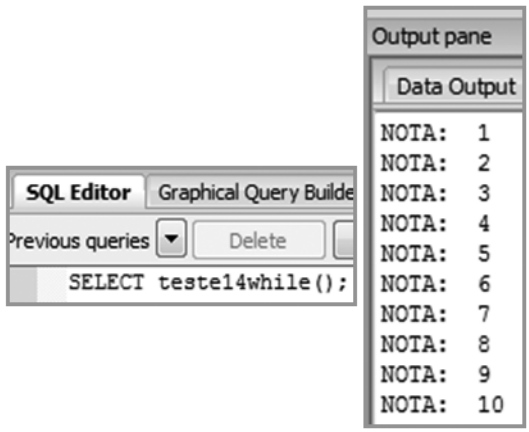


Figura 8.13
Resultado da execução da função teste14lwhile().

Temos também o comando FOR (para), com a sintaxe:

```
FOR <cont> IN [REVERSE] <ini>..<fim> [BY <passo>] LOOP
  -- comandos
END LOOP;
```

O mesmo exemplo onde um bloco de comandos é executado por dez vezes, dessa vez através do uso do FOR, é apresentado na figura 8.14.

```
CREATE OR REPLACE FUNCTION teste15for()
  RETURNS void AS $BODY$
DECLARE i INTEGER;
BEGIN
  FOR i IN 1..10 LOOP
    RAISE NOTICE '%', i;
  END LOOP;
END; $BODY$
LANGUAGE plpgsql;
```

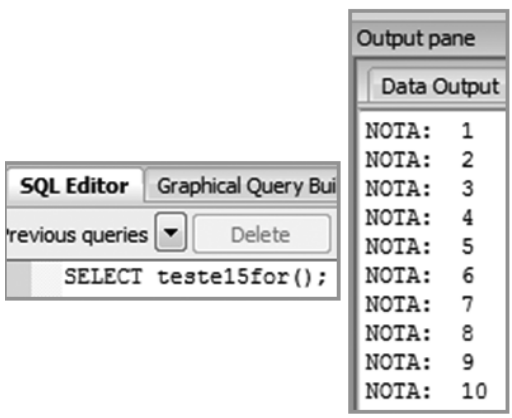


Figura 8.14
Resultado da execução da função teste15for().

Já o exemplo da figura 8.15 ilustra o uso de algumas cláusulas opcionais do FOR, como o REVERSE (o intervalo definido é percorrido em modo decresente) e o BY (que determina o "tamanho" do passo a cada interação). Assim, a variável i receberá inicialmente o valor 40 e, na medida em que a ordenação da repetição está invertida, vai decrescendo de 5 em 5 até atingir o limite final do intervalo. Perceba que a declaração de i está no formato de comentário, já que tal declaração não é necessária.

```
CREATE OR REPLACE FUNCTION teste15reverse()
  RETURNS void AS $BODY$
-- DECLARE i INTEGER;
BEGIN
  FOR i IN REVERSE 40..0 BY 5 LOOP
    RAISE NOTICE '%', i;
  END LOOP;
END; $BODY$
LANGUAGE plpgsql;
```

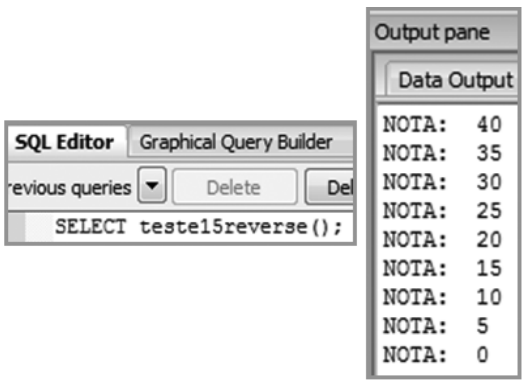


Figura 8.15
Resultado da execução da função teste15reverse().

9

Stored Procedures

objetivos

Aprender sobre o processo de programação em PL/pgSQL para escrita de procedures, function e triggers; Entender o processo de tratamento de erros e uso de cursor no processo de manipulação de dados.

conceitos

Stored procedures; triggers; Tratamento de erros; EXCEPTION; OPEN; FETCH e CLOSE.

Exercício de nivelamento

Você já trabalhou com o tratamento de erros ou exceções durante o desenvolvimento de aplicações? Esse tratamento foi realizado em qual camada?

Tratamento de erros

Como foi visto no capítulo anterior, a linguagem estrutural PL/pgSQL possibilita, entre outras coisa, a implementação de algumas regras de negócios da aplicação em desenvolvimento diretamente no SGBD.

A simples possibilidade de podermos pensar em implementar processamento sobre os dados no servidor traz consigo a necessidade de nos preocuparmos com possíveis erros que possam ocorrer durante a execução de alguma regra de negócio.

Pensando nisso, os projetistas de SGBDs implementaram um conjunto de funcionalidades próprias para que possamos incorporar testes sobre as instruções que serão executadas por uma determinada função e assim evitar interrupções inesperadas, ao mesmo tempo, controlando o fluxo de execução da função de forma a garantir a integridade dos dados.

Por padrão, qualquer *erro* que ocorre em uma função PL/pgSQL interrompe a execução da função e, de fato, da transação envolvida.

Dessa forma, podemos detectar erros e fazer o *tratamento apropriado a partir deles*, usando um bloco BEGIN com uma cláusula de exceção EXCEPTION.

Nesse caso, o processamento dos comandos SQL é abandonado e o controle passa para a lista de exceções, onde os comandos correspondentes são executados e, em seguida, o controle passa para a próxima instrução após END; (fim do bloco de EXCEPTION). A estrutura geral de como isso é programado pode ser vista a seguir:

- Por padrão, qualquer erro que ocorre em uma função PL/pgSQL interrompe a execução da função e, de fato, da transação envolvida
- É possível detectar erros e recuperar a partir deles, usando um bloco BEGIN com uma cláusula de exceção EXCEPTION

```
BEGIN
 <comandosSQL>;
 EXCEPTION
   WHEN <condição> THEN <comandosTratarErro>;
 [ WHEN <condição2> THEN <comandosTratarErro2>; ... ]
END;
```

Se nenhum erro for encontrado na execução do bloco de comandos SQL, o controle do fluxo de execução da função ignora as instruções previstas em EXCEPTION e passa para a próxima instrução após END; (fim do bloco).

O erro se propaga, ou seja, deixar de ser tratado, caso nenhuma correspondência EXCEPTION seja encontrada.

A figura 9.1 trás um exemplo de função, onde foi programado o tratamento de erro (no caso, divisão por zero) através do uso de EXCEPTION:

```
CREATE OR REPLACE FUNCTION teste1exception()
  RETURNS integer AS $BODY$
DECLARE x INTEGER := 10;
BEGIN
  INSERT INTO trabalha (essn, pno, horas)
    VALUES('123456789', '3', 20);
  BEGIN -- try
    UPDATE trabalha SET horas = '35'
      WHERE essn = '123456789' AND pno = '3';
    x := 5 / 0;
    EXCEPTION
      WHEN division_by_zero THEN -- catch
        RAISE NOTICE 'ERRO: divisão por zero';
    RETURN x;  -- finally
  END;
END; $BODY$
LANGUAGE plpgsql;
```

Figura 9.1
Resultado da execução da função teste1exception().

Vejamos os principais passos na execução da função teste1exception() apresentada nesse exemplo:

- Na execução da função, a variável x é inicializada com 10;
- É realizada a inserção na tabela TRABALHA (horas=20) e um novo bloco de comandos é iniciado;
- Para esse sub-bloco, é feita uma tentativa de atualização na tabela TRABALHA (horas=35) e também uma tentativa de divisão por 0;
- Como o segundo comando gera uma exceção, sendo tratada pela cláusula EXCEPTION, os dois comandos anteriores são desfeitos (horas continuará com 20 e x continuará com 10) e a execução é desviada para depois do bloco EXCEPTION.

Cláusula UNIQUE_VIOLATION

A cláusula UNIQUE_VIOLATION é muito utilizada para o tratamento de exceções em operações de INSERT ou UPDATE. Ela permite verificar a violação de unicidade durante a inclusão ou atualização de registros dentro de um banco de dados. Um exemplo de utilização dessa cláusula pode ser visto a seguir:

```
CREATE OR REPLACE FUNCTION set_cliente
  (_nome VARCHAR, _email VARCHAR, _senha VARCHAR )
  RETURNS INTEGER AS $$
DECLARE iIdCliente INTEGER;
BEGIN
  BEGIN
    INSERT INTO cliente (nome, email, senha)
      VALUES (_nome, _email, _senha)
      RETURNING id_cliente INTO iIdCliente;
    EXCEPTION
      WHEN UNIQUE_VIOLATION THEN
        UPDATE cliente SET nome = _nome, senha = _senha
          WHERE email = _email
          RETURNING id_cliente INTO iIdCliente;
```

```
  END;
  RETURN iIdCliente;
END; $$ LANGUAGE 'plpgsql' VOLATILE;
```

Cláusula FOUND

Podemos usar a cláusula FOUND para verificar se uma instrução SELECT retornou ou não registros. No exemplo a seguir, podemos tratar essa condição sem precisar usar a cláusula EXCEPTION, até porque isso implica em menor uso de recursos computacionais.

- Variável especial para verificar retorno no SELECT: **FOUND**

```
Exemplo FOUND: verifica se houve retorno no SELECT.
DECLARE reg RECORD ;
BEGIN
  SELECT * INTO reg FROM categoria WHERE codcat = id;
  IF not found
    THEN RAISE NOTICE 'Categoria % não encontrada', id;
  END IF;
END;
```

- Dica: Um bloco que contém a cláusula EXCEPTION é significativamente mais caro para entrar e sair do que um bloco qualquer. Portanto, não use EXCEPTION sem necessidade

Comando RAISE (levantar)

O comando *RAISE*, já usado no exemplo acima, permite reportar mensagem ou lançar uma exceção. Assim, além de erros que são internamente identificados e tratados pelo PostgreSQL, é possível programar uma situação que equivale a ocorrência de um erro, mas comandada pelo programador.

```
RAISE <Nível Severidade> 'mensagem' [, expressão [,...]];
```

Níveis do comando RAISE:

- DEBUG;
- LOG;
- INFO;
- NOTICE;
- WARNING;
- EXCEPTION.

Entre esses níveis, apenas EXCEPTION cria uma situação equivalente a um erro (que normalmente interrompe a transação corrente).

Na figura 9.2 é apresentada uma função onde o comando *RAISE* é usado tanto para gerar uma exceção (equivale a um erro) como uma mensagem.

```
CREATE OR REPLACE FUNCTION teste3exception(id integer)
  RETURNS void AS $BODY$
DECLARE reg RECORD; i INTEGER := 10;
BEGIN
  SELECT * INTO reg FROM produto WHERE codprod = id;
  IF not found THEN  -- RAISE registro_nao_encontrado;
    RAISE EXCEPTION 'Produto % não existe!', id;
  END IF;
  IF reg.quantprod <= reg.estoqueminprod THEN
    -- RAISE fazer_pedido (escrever em tabela, por ex.)
    RAISE NOTICE 'Produto % com estoque baixo!', id;
  END IF;
  RAISE INFO 'i = %', i;
END;$BODY$
LANGUAGE plpgsql VOLATILE;
```

```
10 – ERRO: Produto 10 não existe!
   ****** Error ******  SQL state: P0001
2 – NOTA: Produto 2 com estoque baixo!
    INFO: i = 10
1– INFO: i = 10
```

Figura 9.2
Exemplo de usos do comando RAISE.

Cursor

É uma variável (área de memória) composta de linhas e colunas que armazena o resultado de uma consulta (zero ou mais registros).

- Normalmente, é preciso seguir os seguintes passos para usar um cursor:
- Declarar variáveis para armazenar os valores das colunas de uma linha;
- Declarar cursor, o qual conterá uma consulta;
- Abrir o cursor;
- Buscar uma linha do cursor por vez e armazenar os valores das colunas nas variáveis declaradas no passo 1, realizando em seguida os processamentos previstos para a função;
- Fechar o cursor após seu uso.

Conforme já apresentado em 5.6, um cursor é uma tabela temporária que pode ser entendida como uma matriz em que cada linha contém um registro e cada coluna corresponde a um campo indicado no comando *SELECT* que deu origem ao cursor.

O interessante é que também é possível declararmos cursores para utilizá-los dentro da lógica de negócio programada através do PL/pgSQL. Um cursor se comporta como uma variável (ocupa área de memória) capaz de armazenar o resultado de uma (consulta de) seleção que retorna 0 (zero) ou mais registros.

A sintaxe a ser observada no uso de cursores pode ser vista a seguir, lembrando que todo cursor deve ser explicitamente declarado na área DECLARE.

```
<nomeC_refcursor> REFCURSOR;
Ou
<nomeC> [[NO]SCROLL] CURSOR [(args)] FOR|IS <QUERY> ;
```

Onde:

- **REFCURSOR**: utilizado para identificar uma variável do tipo cursor;
- **[[NO]SCROLL]**: define o tipo de movimentação do cursor;
- **<QUERY>**: consulta SQL utilizada para criar a tabela temporária para o cursor.

A seguir, exemplos de declaração de cursores:

```
DECLARE
  curs1 refcursor;
  curs2 CURSOR FOR SELECT * FROM empregado;
  curs3 CURSOR (np CHAR) IS SELECT * FROM projeto
                            WHERE pnumero = np;
```

Após declarado, o cursor é manipulado com os comandos *OPEN*, que abre o cursor, *FETCH*, que percorre os registros do cursor, e *CLOSE*, que fecha o cursor. A sintaxe de cada um desses comandos pode ser vista a seguir.

```
OPEN <nomeC_refcursor> [[NO] SCROLL] FOR QUERY;
Ou
OPEN <nomeC> [([arg1 :=] argVlr [, ...])];
FETCH [<direção> {FROM|IN}] <nomeCursor> INTO <reg>;
```

No comando *FETCH*, se não existir a próxima linha, a variável <reg> recebe o valor NULL. A cláusula FOUND pode ser utilizada para verificar se uma linha foi retornada ou não. Não é preciso especificar <direção>, mas nesse caso o padrão é considerar que a direção especificada é NEXT. Outras possibilidades para <direção> são: pRIOR, FIRST, LAST, ABSOLUTE n, RELATIVE n, FORWARD ou BACKWARD (cursor aberto ou declarado com SCROLL).

```
CLOSE <cursor>; -- libera memória
```

Na figura 9.3 é apresentado exemplo de uma função onde um cursor é percorrido através do LOOP até que a condição NOT FOUND (fim do cursor) seja verdadeira, causando a saída do LOOP através do EXIT.

```
CREATE OR REPLACE FUNCTION teste1cursor(ndepto character)
  RETURNS void AS $BODY$
DECLARE c_empPorDepto CURSOR IS
          SELECT * FROM empregado WHERE dno = $1;
                                        -- $1 ref. para ndepto
        reg RECORD;
BEGIN
  OPEN c_empPorDepto;
  LOOP
    FETCH c_empPorDepto INTO reg;
    IF NOT FOUND
      THEN EXIT;
    END IF;
    RAISE NOTICE 'Empregado % %. %', reg.pnome,
            reg.inicialm, reg.unome;
  END LOOP;
  CLOSE c_empPorDepto;
END; $BODY$  LANGUAGE plpgsql;
```

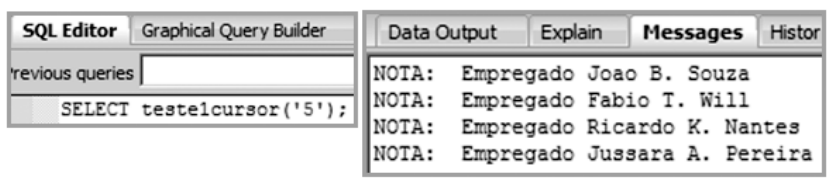


Figura 9.3
Resultado da execução da função teste1cursor().

O PostgreSQL disponibiliza uma forma alternativa para a instrução FOR que permite iteração nas linhas retornadas por um cursor. Essa estrutura encapsula as ações de abertura, movimentação e fechamento do cursor, deixando o código da função menor e mais legível. A sintaxe desse FOR é:

```
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ]
argument_value [, ...] ) ] LOOP
    <statements>
END LOOP
```

Na figura 9.4, é mostrado um exemplo utilizando essa alternative de FOR.

```
CREATE OR REPLACE FUNCTION teste2cursor(ndepto character)
  RETURNS void AS $BODY$
DECLARE c_empPorDepto CURSOR (id character) IS
                SELECT * FROM empregado WHERE dno = id;
        reg RECORD;
BEGIN
  -- abre cursor, carrega registro no 'contador' e
  -- fecha cursor
  FOR reg IN c_empPorDepto(ndepto) LOOP
    RAISE NOTICE 'Emp: % %. %', reg.pnome,
                 reg.inicialm, reg.unome;
  END LOOP;
END;
$BODY$
LANGUAGE plpgsql;
```

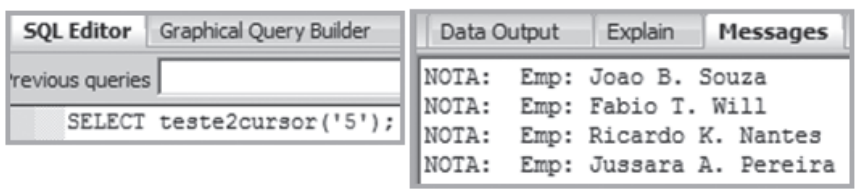


Figura 9.4
Resultado da execução da função teste2cursor().

Outro exemplo, apresentado na figura 9.5, realiza a classificação de alunos de acordo com a nota (desempate pela idade). Nota 0 é desclassificada.

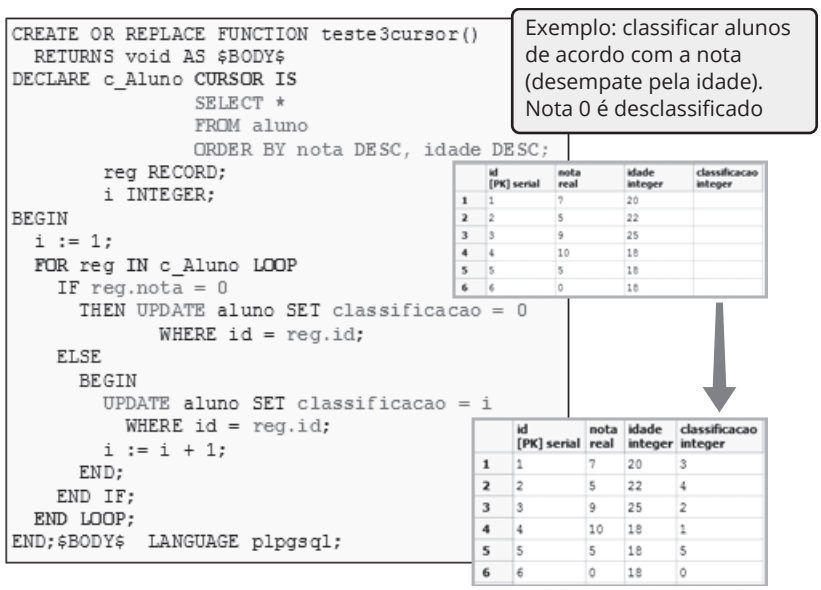


Figura 9.5
Resultado da execução da função teste3cursor().

Atividade de Prática AVA

Trigger

- Composição de um trigger:
- Momento: BEFORE (antes), AFTER (depois) ou INSTEAD OF (ao invés de).
- Evento(s): INSERT e/ou DELETE e/ou UPDATE.
- Tabela: nome da tabela afetada pelo trigger.
- Tipo: STATEMENT (default) ou ROW: indica se a trigger será executada uma única vez (considerando o STATEMENT como um todo) ou se será executada para cada linha (ROW) afetada pela instrução que disparou o gatilho.
- Corpo: é o bloco de código (PL/pgSQL).

Um trigger (gatilho) é uma procedure executada (ou disparada) automaticamente pelo banco de dados, quando uma instrução DML (INSERT, UPDATE OU DELETE) é executada em uma tabela predeterminada. Triggers podem ser disparados antes ou depois da execução de uma instrução DML.

Geralmente são aplicadas para garantir restrições de integridade complexas e que não podem ser geradas na criação das tabelas, para registrar informações diversas sobre modificações de tabelas e para sinalizar outros programas sobre modificações ocorridas.

A definição do momento de execução de um trigger deve considerar que o BEFORE faz com que esta seja executada antes de modificar os dados, sendo por isso mesmo indicado para inicializar variáveis globais, validar regras de negócio, alterar o valor de flags ou salvar o valor da coluna antes de modificar seu valor. Já o AFTER faz com que a trigger seja executada somente depois que os dados tenham sido modificados, sendo normalmente utilizado quando é necessário completar dados em outras tabelas do SGBD. O INSTEAD OF, por sua vez, faz com que a instrução que o disparou seja substituída pelo próprio trigger.

Uma função (procedure) de gatilho é criada com o comando *CREATE FUNCTION*, declarando-a como uma função sem argumentos e com um tipo de retorno TRIGGER, conforme o exemplo a seguir.

```
CREATE OR REPLACE FUNCTION <nomeTriggerFunction>
  RETURNS trigger AS $BODY$
BEGIN
  <comandos>;
END; $BODY$
LANGUAGE plpgsql;
```

Quando uma função de gatilho é executada, variáveis especiais são automaticamente criadas, como por exemplo:

- **OLD**: do tipo RECORD, contém o valor anterior (antigo) da coluna no banco de dados em operações de UPDATE/DELETE;
- **NEW**: do tipo RECORD, contém o novo valor da coluna no banco de dados em operações de UPDATE/INSERT;
- **TG_OP**: do tipo TEXT, contém a operação que disparou o gatilho (INSERT, UPDATE, DELETE).

A sintaxe para criação da função a ser executada pelo trigger é a seguinte:

```
CREATE OR REPLACE TRIGGER <nomeTrigger>
{BEFORE|AFTER}
{DELETE OR INSERT OR UPDATE [OF <coluna>]} ON <nomeTab>
{FOR EACH ROW|STATEMENT}
[WHEN] -- Restringe linhas que dispararão o gatilho
EXECUTE PROCEDURE < nomeTriggerFunction >;
```

A figura 9.6 mostra a janela que é exibida pelo assistente do pgAdmin3 ao acionar a criação de um novo trigger.

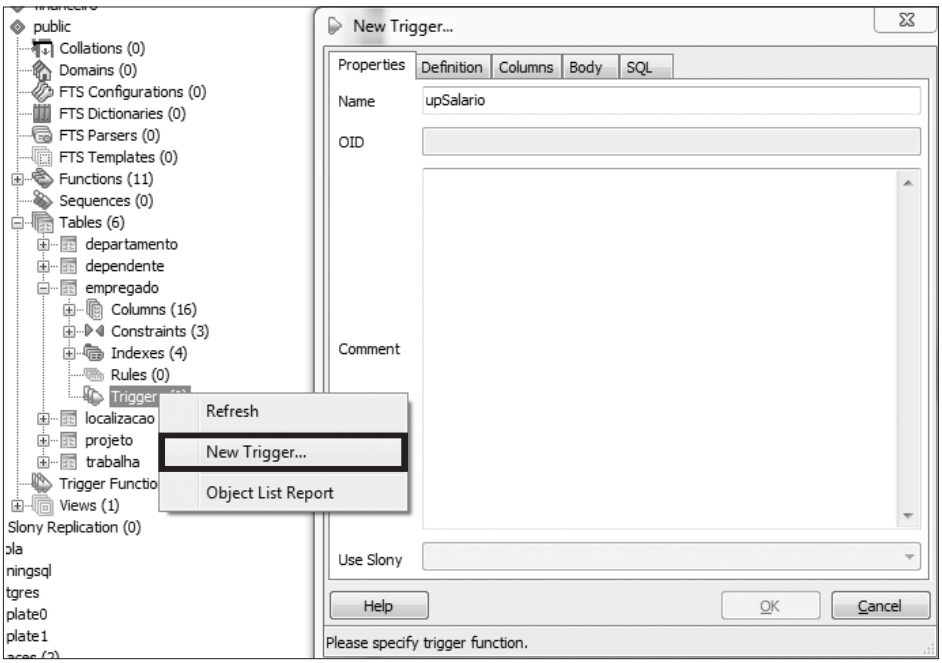


Figura 9.6
Criação de Trigger através do assistente pgAdmin3.

Como exemplo de restrição de integridade a ser verificada, temos um trigger upSalario que será aplicado para cada registro (ROW) que vier a ser atualizado na tabela empregado:

```
CREATE TRIGGER "upSalario"
  BEFORE
  UPDATE OF salario ON empregado
  FOR EACH ROW
  EXECUTE PROCEDURE "upSalarioEmp"();
```

A função "upSalarioEmp"(), por sua vez, será executada pela trigger upSalario antes de consolidar a instrução SQL UPDATE, fazendo a verificação se o novo salário é menor que o velho, antes de proceder com a atualização, conforme podemos ver a seguir:

```
CREATE OR REPLACE FUNCTION “upSalarioEmp”()
  RETURNS trigger AS $BODY$
BEGIN
   IF NEW.salario < OLD.salario
      THEN RAISE NOTICE 'Novo salário menor que antigo';
   END IF;
   RETURN NEW;
END; $BODY$
LANGUAGE plpgsql;
```

Nesse outro exemplo, envolvendo um campo derivado, o total de empregados por departamento (na tabela DEPARTAMENTO) é atualizado durante o processo de manutenção (INSERT, UPDATE, DELETE) de registros na tabela EMPREGADO. Assim, temos uma trigger manEmp:

```
CREATE TRIGGER “manEmp”
  AFTER INSERT OR UPDATE OR DELETE  ON empregado
  FOR EACH ROW EXECUTE PROCEDURE “manipulaEmp”();
```

E a função manipulaEmp que será executada pela trigger manEmp antes de consolidar a instrução SQL INSERT/UPDATE/DELETE.

```
CREATE OR REPLACE FUNCTION “manipulaEmp”()
  RETURNS trigger AS $BODY$
BEGIN
  IF TG_OP = 'INSERT' THEN
    UPDATE departamento SET numemp = numemp + 1
      WHERE dnumero = NEW.dno;
  ELSIF TG_OP = 'DELETE' THEN
       UPDATE departamento SET numemp = numemp - 1
         WHERE dnumero = OLD.dno;
     ELSIF TG_OP = 'UPDATE' THEN
          UPDATE departamento SET numemp = numemp - 1
            WHERE dnumero = OLD.dno;
          UPDATE departamento SET numemp = numemp + 1
            WHERE dnumero = NEW.dno;
  END IF;
  RETURN null;
END; $BODY$ LANGUAGE plpgsql;
```

Teste: inserir um registro na tabela empregado

```
INSERT INTO empregado(ssn,pnome,unome, sexo,dno, salario)
   VALUES ('111111111','CARLA','SILVA','F','1',55000);
```

Exercício de Fixação 

Quais são três possíveis categorias que podem ser indicadas quando da criação de uma função em PostgreSQL? O que os diferenciam?

10

Transações

objetivos

Conhecer, entender o contexto e ver exemplos do processo de controle de transações concorrentes no Banco de Dados.

conceitos

Transações; Concorrência; Bloqueio (lock); Impasse (dead-lock); Semáforo; MVCC; ACID; COMMIT; ROLLBACK; log e savepoint.

Exercício de nivelamento

Você já desenvolveu alguma aplicação em que teve de se preocupar com um grande número de usuários executando transações concorrentes no Banco de Dados? Se sim, cite um processo onde foi necessário realizar tal controle.

Concorrência

Qualquer banco de dados geralmente prevê uso por um grupo considerável de usuários. Isso levanta preocupações com o processo de administração e controle de concorrência das informações que serão manipuladas, necessários quando existe a possibilidade de usuários distintos tentarem acessar a mesma informação ao mesmo tempo. Nesses casos, é preciso garantir a integridade das informações, por exemplo, cuidando para não permitir duas vendas de um único produto em estoque.

Geralmente temos como cenário um SGBD com grande volume de informações nele armazenado e uma quantidade considerável de usuários manipulando essas informações. Exemplos disso são sistemas de reserva de passagens, bancos, processamento de cartões de crédito, mercado de ações, supermercados etc. Em todos eles os usuários querem sempre:

- Alta disponibilidade;
- Baixo tempo de resposta;
- Confiabilidade.

O problema é que sistemas computacionais são complexos e podem existir falhas em sua operação, sendo necessário lidar com situações inesperadas.

Até agora nos preocupamos somente com a execução de instruções SQL de forma isolada, como se um único usuário pudesse ter todo o banco de dados somente para ele. Porém, em BD multiusuários, instruções SQL são concorrentes.

- Exemplo: consultar saldo de todas as C/C de uma agência;
- Múltiplos usuários podem:
  - Fazer depósitos;
  - Sacar dinheiro.
- O próprio sistema pode lançar "rendimentos" sobre o saldo em conta;
- Quais serão os saldos apresentados no relatório final?
- Depende de como o SGBD trata o bloqueio de dados: lOCKING

Importante: em SO, acesso concorrente ao dado: sofre Condição de corrida que pode levar a Inconsistência do dado.

- Tratamento: problema da Seção Crítica;
- Solução: sincronização (semáforo: WAIT e SIGNAL).

Uma possibilidade de implementação de controle é por meio de multiprogramação. A ideia é executar alguns comandos de um processo, depois suspender esse processo e executar alguns comandos do próximo processo, sendo que de tempos em tempos o processo que foi suspenso deve ser retomado do ponto em que foi suspenso. Isso faz com que tenhamos uma execução intercalada com o controle de concorrência dos processos acessando uma determinada informação no banco de dados.

Na figura 10.1, a seguir temos uma simulação onde dois processos concorrentes estão reservando lugares em um voo qualquer.

Tempo	Sra. Maria	Sr. Eduardo
13:45	Lê registro de voo e encontra **25** lugares disponíveis	
13:48		Lê registro de voo e encontra **25** lugares disponíveis
13:52	Deduz **4** lugares, grava registro do voo atualizado, indicando **21** lugares disponíveis	
13:56		Deduz **6** lugares, grava registro do voo atualizado, indicando **19** lugares disponíveis

Porém, neste ponto, o registro de voo deveria mostrar **15** lugares disponíveis!

A figura descreve uma situação onde a estratégia de multiprogramação parece não funcionar. Na verdade, existem diversas técnicas de controle de concorrência que podem ser utilizadas como forma de assegurar a propriedade de não interferência entre uma operação e outra, ou o isolamento das transações executadas ao mesmo tempo.

Figura 10.1
Acesso concorrente a informações de reserva de passagens.

Grande parte dessas técnicas garante a serialização, que é a execução das transações de forma serial (uma atrás da outra). Para isso, é necessário saber quais são todas as operações executadas entre o início e o fim de uma transação. No exemplo da figura 10.1 podemos supor que a transação de reserva engloba os comandos de consultar ao número de lugares disponíveis, verificar se a quantidade disponível é igual ou maior do que a quantidade de lugares desejados e, finalmente, subtrair o total de assentos reservados do estoque de lugares disponíveis. Mais à frente, vamos detalhar melhor o conceito de transações em um SGBD.

Bloqueios

Locking ou bloqueio é o mecanismo que o SGBD usa para controlar o acesso simultâneo aos dados nele armazenados. Esse controle pode ser obtido por meio de duas estratégias distintas, que dependem das operações básicas de leitura e gravação (r/w). São elas:

- 1 Write lock / n Read lock.
  - Mais restritivo. Exemplo: problema escritor/leitores.
- Write lock.
  - Geração de versão para leitor no início (versionamento).

Em ambos os casos, o tempo de espera pode ser longo.

Granularidade dos Bloqueios

Outro aspecto a ser considerado em bloqueios é sua granularidade, ou "nível" dos objetos ou componentes que podem ser bloqueados, impedindo assim que múltiplos usuários acessem ou modifiquem um determinado dado no nível apropriado.

Os possíveis níveis de bloqueio são:

- Banco de dados;
- Tabela (table locks): a mesma tabela simultaneamente;
- Página (page locks): a mesma página de uma tabela:
  - Página: segmento de memória entre 2 a 16 KB.
- Linha (row locks): a mesma linha de uma tabela;
- Campo.

O Postgre SQL, assim como o Oracle, implementa o bloqueio de linha (row lock). Apesar do bloqueio de tabela exigir poucos recursos computacionais, ele impede que outros usuários acessem qualquer dado naquela tabela até que a mesma seja liberada. Em um ambiente com muitos usuários, isso acaba resultando em um tempo de espera que é inaceitável. Assim, o bloqueio por linha acaba por exigir mais recursos computacionais, mas permite que mais usuários possam modificar simultaneamente recursos de uma mesma tabela (desde que em linhas diferentes).

O resultado final é que no bloqueio por linha o tempo de espera quando dois usuários querem acessar uma mesma linha será muito menor se comparado com o tempo que teriam de esperar por todos os demais usuários (mesmo que estivessem acessando outras linhas).

MVCC

O PosgreSQL utiliza o modelo de bloqueio multiversão Multiversion Concurrency Control (MVCC), que permite que uma leitura não bloqueie uma escrita e nem que uma escrita bloqueie uma leitura. É um método ainda mais "caro" em termos computacionais, já que o SGBD cria uma versão dos dados acessados (snapshot) para ser utilizada por uma transação qualquer, evitando assim o problema de uma transação acessar dados inconsistentes.

Dessa forma, o PostgreSQL consegue gerenciar a concorrência entre transações de forma automática, mas existem limitações que devem ser consideradas. De qualquer modo, é possível "desligar o piloto automático" e estabelecer granularidade de bloqueio de tabela ou linha em situações onde o MVCC não for a melhor alternativa.

Saiba mais

Mais à frente, mostraremos como lidar com essas questões.

Bloqueios e impasses

Em algumas situações, o uso de bloqueios pode levar a um impasse, também conhecido como deadlock, conforme demonstrado na figura 10.2.

Tempo	Sra. Joana	Sr. Pedro
10:15	Obtém e bloqueia registro de **porcas**	
10:16		Obtém e bloqueia registro de **parafusos**
10:17	Tenta obter (e bloquear) registro de parafusos, mas está bloqueado ao Sr. Pedro	
10:18		Tenta obter (e bloquear) registro de porcas, mas está bloqueado a Sra. Joana

DEADLOCK!

Figura 10.2
Acesso concorrente com geração de impasse (deadlock).

Uma possível solução para esse impasse é a aloção total dos recursos para um dos usuários, sendo que os demais só poderão utilizá-los após a conclusão das operações sendo realizadas pelo primeiro usuário. Mas para isso funcionar, é necessário implementar mecanismos de monitoramento pelo SGBD que consigam identificar a ocorrência de situações de impasse e atuar para resolvê-las.

Uma alternativa é o uso de versionamento em conjunto com mecanismos de monitoramento. Nesse caso, são fornecidas cópias dos recursos (dados) para cada transação de cada cliente, sendo de responsabilidade do monitor lidar com possíveis conflitos. Em geral, a transação mais antiga é priorizada e as demais são reiniciadas em seguida.

Transações

Até agora, o termo transação foi referenciado mais de uma vez, sem, no entanto, ter sido claramente definido. Uma transação em um SGBD é vista como um programa em execução que inclui uma ou mais operações SQL.

Essas transações podem estar embutidas em um módulo ou sistema acessando o SGBD através de conexões cliente, ou podem ser definidas através de uma ferramenta com interface gráfica amigável, como é o caso do pgAdminIII já utilizado nesse curso. Através dessas ferramentas, podem ser criadas e executadas transações para manipular os dados armazenados em um banco de dados qualquer.

O mais importante, contudo, é estabelecer os limites da transação, ou seja, definir exatamente seu inicio e fim. Por conta disso, podemos entender uma transação como uma unidade atômica de trabalho que, ou é realizada com sucesso, ou simplesmente não é realizada, mas que jamais será executada somente em parte.

Transações são uma forma de dar suporte às operações concorrentes, garantindo a segurança e integridade dos dados, apresentando um conjunto de propriedades que forma a sigla ACID. São elas:

- **Atomicidade**: indivisível (todas as operações são executadas ou nada acontece);
- **Consistência**: transações não podem quebrar as regras do BD (manter consistência);
- **Isolamento**: transações simultâneas não sofrem interferências umas das outras;
- **Durabilidade/permanência**: efeitos de uma transação de sucesso são persistidos no BD (mesmo em presença de falhas).

Conforme já visto na sessão 4, um SGBD em geral dispõem de uma DTL, que é a linguagem de Controle de Transações e que fornece mecanismos para controlar transações no banco de dados. No exemplo a seguir veremos como os comandos *COMMIT* e *ROLLBACK* são utilizados.

Exemplo: transferir R$ 500,00 da conta corrente para conta poupança. Durante a transferência, expirou bloqueio ou o servidor desligou.

Etapas:

1. Inicia
2. Invoca instruções SQL
3. Se algo errado, ROLLBACK (desfaz os comandos, abortando a operação como um todo). Se não, COMMIT (efetiva a transação – operações executadas no banco de dados).

Vejamos agora, na figura 10.3, como a transação do exemplo acima é implementada com comandos SQL.

```
START TRANSACTION;
/*
     saca dinheiro da primeira conta, certificando-se de que o
saldo é suficiente */
UPDATE conta SET saldo = saldo - 500 WHERE numero = 1 AND saldo >
500;
IF <exatamente uma linha foi atualizada pela instrução anterior>
THEN
     /* deposita dinheiro na segunda conta /*
     UPDATE conta SET saldo = saldo + 500 WHERE numero = 2;
     IF <exatamente uma linha foi atualizada pela instrução
anterior> THEN
      /* tudo ocorreu bem /*
      COMMIT;
     ELSE
      /*algo deu errado, desfaça todas as mudanças dessa
transação*/
      ROLLBACK;
     END IF;
ELSE
     /*fundos insuficientes ou erro encontrado durante a
atualização*/
     ROLLBACK;
END IF;
```

Figura 10.3
Controle de transação no banco de dados.

De um modo geral, temos a seguinte sintaxe para uma transação:

```
START TRANSACTION / BEGIN [WORK|TRANSACTION];
<comandos>;
COMMIT [WORK|TRANSACTION];
ou
ROLLBACK [WORK|TRANSACTION];
```

Já foi mencionado, contudo, que apesar do PostgreSQL conseguir gerenciar a concorrência entre transações de forma automática, existem situações em que isso pode ocasionar problemas. Nesses casos, o mais indicado é desativar esse recurso, conhecido como autoefetivação (autocommit) e assumirmos a tarefa de realização do controle de transações por conta própria.

Considere, por exemplo, uma transação de atualização de saldo de uma conta dentro de uma tabela do banco de dados:

```
CREATE TABLE conta( numero integer NOT NULL,  saldo real,
   CONSTRAINT conta_pkey PRIMARY KEY (numero));

INSERT INTO conta VALUES (1, 500);
```

Nessa situação, pode ser necessária a implementação de controle de transação para possibilitar a correta atualização dos dados de um determinado registro durante a execução de instruções SQL em processos de interfaces diferentes que necessitam atualizar ou consultar dados de um mesmo registro, conforme ilustrado na figura 10.4.

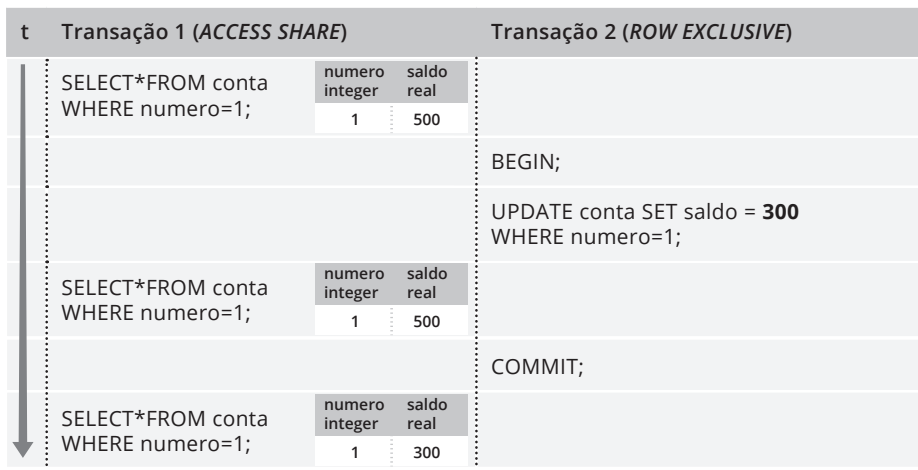


Figura 10.4
Fluxo de execução de transação sobre registro na tabela conta.

O mais indicado seria desativar o modo autocommit de forma permanente e se habituar a executar todas as suas operações SQL dentro de uma transação. Dessa forma, a verificação do estado final do banco de dados após a execução de uma instrução SQL passa ser obrigatória para qualquer uma das transações realizadas.

Bloqueios no PostgreSQL

De modo a permitir o controle de concorrência, o PostgreSQL reconhece oito tipos de bloqueios:

- ACCESS SHARE
- ROW SHARE
- ROW EXCLUSIVE
- SHARE UPDATE EXCLUSIVE
- SHARE
- SHARE ROW EXCLUSIVE
- EXCLUSIVE
- ACCESS EXCLUSIVE.

Sintaxe:

```
LOCK [ TABLE ] [ ONLY ] name [,...] [ IN lockmode MODE ] [ NOWAIT ]
```

Onde LOCKMODE pode ser:

- **Access Share:** bloqueio requerido apenas por instruções SELECT e ANALYZE são simples leitura em geral permitidas, a menos que alguma outra transação tenha requerido Access Exclusive;
- **Row Share:** a instrução SELECT com a cláusula FOR UPDATE obtém esse nível de bloqueio. Ele conflita apenas com os níveis mais retritos Exclusive e Access Exclusive;
- **Row Exclusive:** bloqueio solicitado automaticamente por instruções INSERT, UPDATE, DELETE. Conflita com quem precisar, além de Access Exclusive e Exclusive, também com Share Row Exclusive e Share;
- **Share Update Exclusive:** bloqueio padrão do comando VACUUM sem a opção FULL. Conflita com todos os próximos níveis de bloqueio e com ele próprio. Bloqueia a tabela inteira;
- **Share:** bloqueia toda a tabela e é requerido por instruções CREATE INDEXEMPLO:
- **Share Row Exclusive:** similar ao Exclusive (ver a seguir), mas apenas no nível de linha. Não há instrução SQL que automaticamente solicite esse nível de bloqueio;
- **Exclusive:** esse só não conflita com Access Share. É requerido automaticamente apenas em tabelas de catálogo em determinadas operações;
- **Access Exclusive:** conflita com todo mundo. Ou seja, nenhuma outra transação pode operar na mesma tabela bloqueada por alguma transação que solicitou Access Exclusive. Instruções ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER e VACUUM FULL necessitam desse modo de bloqueio.

Vejamos a seguir, nas figuras 10.5 e 10.6, mais alguns exemplos de transações no PostgreSQL e os respectivos tipos de bloqueio por elas utilizados.

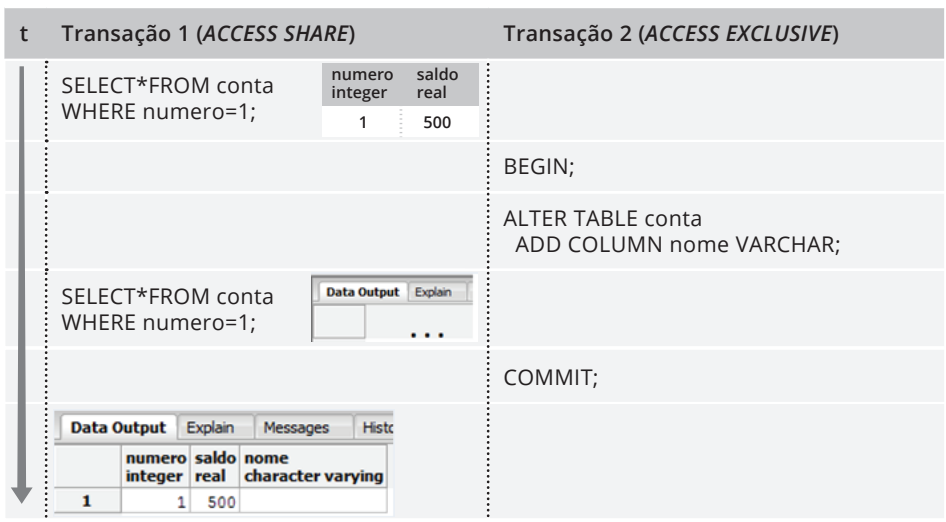


Figura 10.5
Fluxo de execução de transação de atualização de tabela.

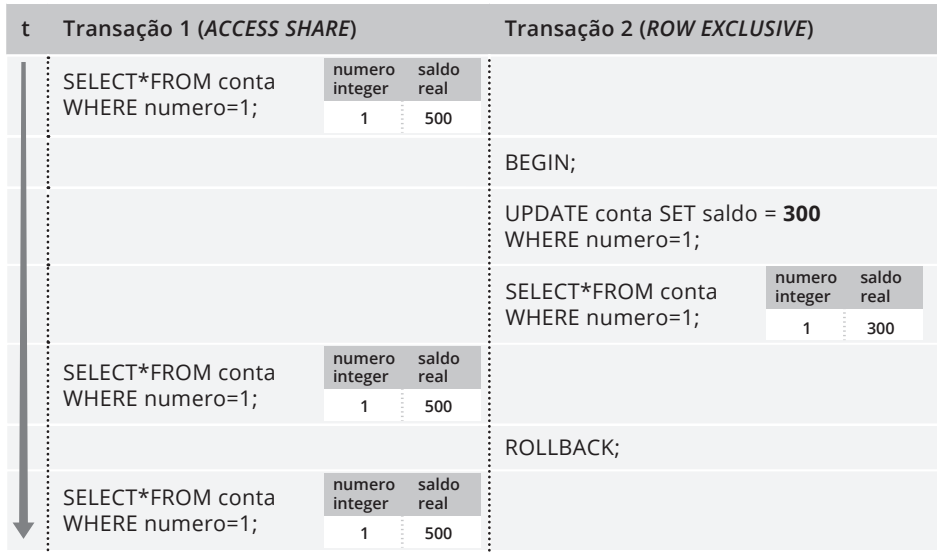


Figura 10.6
Fluxo de execução de transação que não se concretiza.

Log e Savepoints

Os SGBDs mantêm um log de transações (o diário do SGBD) para controlar todas as operações da transação que afetem valores armazenados no banco de dados.
No PostgreSQL ele é chamado Write Ahead Log (WAL).

Características do WAL são:

- Entradas:
  - [start_transaticon T]
  - [write, T, X, valor_antigo, novo_valor]
  - [read, T, X]
  - [commit, T]
  - [abort, T]
- É armazenado em disco (buffer na MP), no diretório pg_xlog;
- Recursos alocados são liberados quando a transação é concluída (desfeita ou executada);
- Permite a restauração de falhas. Se servidor desligado:
  - Antes do COMMIT/ROLLBACK: quando ficar on-line, deve desfazer transações incompletas;
  - Durante COMMIT: reaplicar modificações registradas no log (propriedade de DURABILIDADE).

Já os Savepoints são pontos dentro de uma transação que indicam que os comandos posteriores podem sofrer rollback, enquanto os comandos anteriores são mantidos no banco de dados mesmo que a transação tenha sido abortada. Uma transação pode ter mais de um Savepoint, que é indicado por um nome fornecido pelo programador.

No PostgreSQL, este recurso é chamado de Point In Time Recovery (PITR), e permite voltar a um ponto específico no tempo (por meio do WAL). Existem comandos específicos para indicar um SAVEPOINT, que deve ter sempre um nome. Esses comandos estão resumidos nas tabelas a seguir:

Opção	Descrição
SAVEPOINT *nome*	Registra um ponto de passagem na transação que poderá ser utilizado pela posteriormente.
RELEASE SAVEPOINT *nome*	Elimina um determinado ponto de passagem que tenha sido criado anteriormente à sua chamada.
ROOLBACK TO SAVEPOINT *nome*	Retorna a transação até o ponto de passagem informado.

Tabela 10.1
Comandos específicos para indicar um SAVEPOINT.

Para concluir, apresentamos um exemplo fazendo uso de um SAVEPOINT:

```
-- início da transação

BEGIN;

-- considerando a existência do registro (1, 500)

INSERT INTO conta VALUES(2, 500);

SAVEPOINT aqui;

INSERT INTO conta VALUES(3, 1000);

SELECT * FROM conta; -- são mostrados os registros 1, 2 e 3

DELETE FROM conta;

SELECT * FROM conta; -- nada é mostrado

ROLLBACK TO aqui;

SELECT * FROM conta; -- são mostrados os registros 1 e 2

ROLLBACK;

SELECT * FROM conta; -- é mostrado o registro 1

-- fim da transação
```

Exercício de Fixação
Transações

Descreva as funcionalidades presentes no acrônimo ACID.

Paulo Henrique Cayres possui graduação no curso Superior de Tecnologia em Processamento de Dados pela Universidade para o Desenvolvimento do Estado e da Região do Pantanal (UNIDERP), especialização em Análise de Sistemas pela Universidade Federal de Mato Grosso do Sul (UFMS) e mestrado em Ciências da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS). Atuou como coordenador curso de Bel. em Sistemas de Informação e Superior de Tecnologia em Redes de Computadores na Faculdade da Indústria do Sistema FIEP, onde também coordenou as atividades do SGI - Setor de Gestão de Informações. Atualmente é coordenador do Núcleo de Educação a Distância - NEaD da Faculdade da Indústria do Sistema FIEP. Sócio-diretor da CPP Consultoria e Assessoria em informática Ltda. Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: linguagens de programação, engenharia de software, modelagem de sistemas, desenvolvimento de aplicações para web e gerência de projetos. Professor titular em cursos de graduação e pós-graduação ministrando disciplinas de desenvolvimento de sistemas desde 1995. Instrutor de treinamento na linguagem Java de programação junto ao CITS em Curitiba e na ESR-RNP.

esr.rnp.br

LIVRO DE APOIO AO CURSO

O curso apresenta uma visão geral sobre bancos de dados, bem como conceitos e metodologias para modelagem conceitual, lógica e física de banco de dados relacionais. Aborda as principais características e funcionalidades de um sistema gerenciador de bancos de dados usando como base o PostgreSQL. Explora de forma prática a criação e manutenção de bases de dados bem como a pesquisa de informações nelas armazenadas através da linguagem SQL.

ISBN 978-85-63630-50-6
9 788563 630506