

Universidade Estadual de Campinas
Colégio Técnico de Campinas
Departamento de Processamento de Dados



Desenvolvimento de Sistemas
Informática

Bancos de Dados



2023

Material compilado e organizado por
Prof. Francisco da Fonseca Rodrigues

Sumário

1 Para que serve um Banco de Dados	4
2 Definições Iniciais	6
Dado, Informação e Metadados	6
Bancos de Dados	7
Sistemas de Gerenciamento de Banco de Dados	8
Tipos de usuários	10
Características e Funcionalidades	10
Aplicações de Bancos de Dados	10
Tipos de Bancos de Dados e sua evolução	11
3 Modelagem	15
Elementos da Modelagem Conceitual (entidades, relacionamentos, atributos, tipos)	15
Fases da Modelagem Conceitual	17
Análise de Requisitos e Modelo Conceitual	17
Modelo Lógico: Entidade, Registro, Campo, Relacionamento, Chaves	19
Diagrama Entidade-Relacionamento	20
Cardinalidade e tipos de Relacionamento	29
Restrições de Integridade	37
Dicionário de Dados	40
Normalização: Anomalias, Dependências Funcionais, Formas Normais	65
Modelo Físico: instalação e uso do SSMS	65
4 SQL - Linguagem de Consulta Estruturada	69
Tipos de dados de atributos	72
Criação de Banco de Dados, Esquema, Tabelas e Relacionamentos no servidor	75
Diagrama de Banco de Dados no SSMS	84
Comandos de Manipulação de dados: Insert, Alter Table, Update, Delete	87
Interação dos comandos de manipulação de dados com restrições de integridade	98
Comando de Consulta a dados: Select, From, condições de seleção e de junção, Alias	100
Tabelas vistas como conjuntos: Distinct, Union, Like, Order By, Top	107
Consultas Avançadas: null, Exists, Consultas aninhadas, IN, Between	117
Junções de tabelas: interna, externa, completa e cruzada	124
Funções de agregação e Agrupamento de registros	131
5 Programação de aplicativo com acesso direto a servidor de banco de dados	145
6 Otimização de consultas: Check, Views e Index	172
7 Programação no Servidor de Banco de Dados	183
Lotes de Comandos	183
Declaração de Variáveis	183
Comandos de Controle de Fluxo de Execução	184
Triggers	186
Stored Procedures	189
Tratamento de Exceções	191
SQL Injection	196
Cursores e Transações	203
8 Programação de Aplicações em Camadas	210
9 Conceitos de Bancos de Dados não-relacionais (NoSQL) – a fazer	276
10 Aspectos legais e éticos do uso de Sistemas de Bancos de Dados - a fazer	

Material baseado em

- Guia Mangá de Bancos de Dados
- www.bosontreinamentos.com.br
- Apostila Modelagem de Bancos de Dados – prof. Marcos Alexandruk

Slides para estudo complementar:

<https://slideplayer.com.br/slide/1263574/>

sites.google.com/site/uniplibancodedados1/aulas/

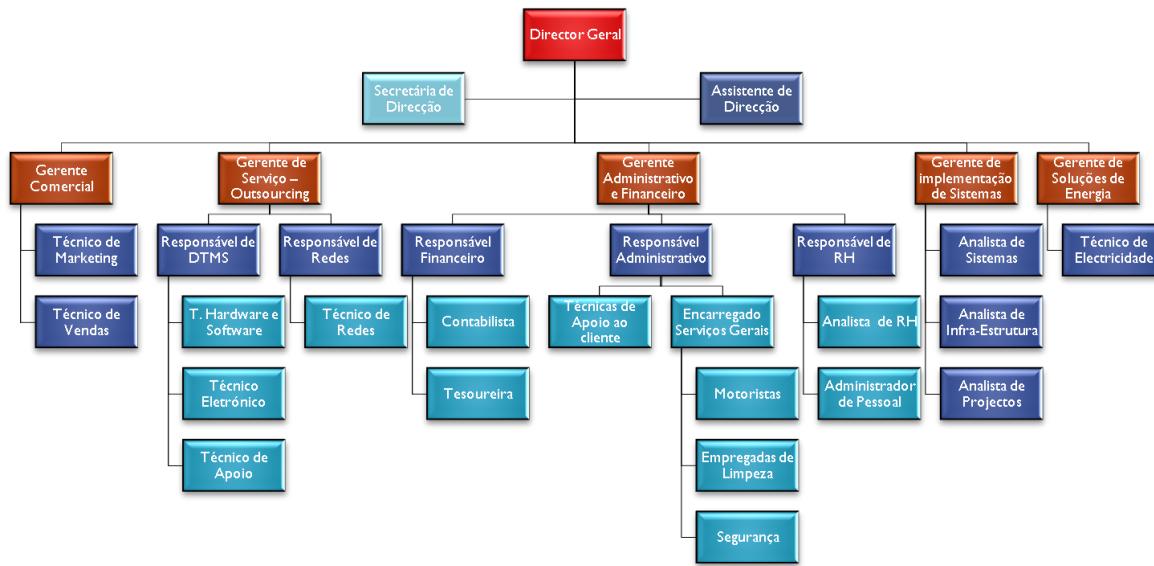
Vídeos para estudo complementar:

- Kim, Won. *Introduction to Object-Oriented Databases*. The MIT Press, 1990. [ISBN 0-262-11124-1](#)
- <https://www.lucidchart.com/pages/pt/simbolos-de-diagramas-entidade-relacionamento>
- <http://docente.ifrn.edu.br/abrahaolopes/sementre-2013.1/3.2401.1m-prog-bd/>

1. Para que serve um Banco de Dados

Imagine uma empresa aérea, como a Azul, Latam ou Aeromexico. Essas empresas são compostas por vários setores, como o Setor de Manutenção, o Setor de Reserva de Passagem, o setor de Recursos Humanos, o Setor de Finanças, Setor de Vendas, Setor de Compras, Setor de Treinamento, Setor de Segurança, Setor de Informática, dentre muitos outros. Isso ocorre porque uma empresa aérea é um empreendimento de alta complexidade.

Muitas empresas possuem setores, que trabalham em operações específicas, mas que precisam, geralmente, interagir entre si. Abaixo temos um exemplo de estrutura de uma empresa qualquer:



Voltando às empresas de aviação, para desempenhar suas funções, elas precisam trabalhar com uma quantidade enorme de informações, como passageiros, voos, aviões, pilotos, rotas aéreas, aeroportos, fornecimento de alimentos, vestuário, manutenção de equipamentos, dentre muitas outras.

Imagine que cada setor da empresa aérea tenha cópias dos arquivos com as informações necessárias para seu funcionamento. Muitas dessas informações serão repetidas entre os vários setores. Por exemplo, o setor de Finanças precisa saber as informações dos pilotos e tripulantes para poder efetuar o pagamento de seus salários. O setor de Recursos Humanos também precisa de informações semelhantes para alojar os pilotos e tripulantes nos diversos voos. Mas, para saber quais são os voos, precisa de dados iguais aos do Setor de Reserva de Passagem. O Setor de Vendas também precisa dessas informações, quando efetuar o recebimento dos valores pagos pelas passagens. O Setor de Treinamento precisa saber quem são os pilotos e tripulantes, para poder matricular-lhos em treinamentos. Além de tudo isso, o Setor de Qualidade precisa de todas essas informações para averiguar se o funcionamento da empresa aérea está dentro dos parâmetros de qualidade estabelecidos para que, caso não esteja, indicar ações de melhoria que poderão envolver, por exemplo, o Setor de Segurança.

Se cada um desses setores tiver seus próprios arquivos contendo essas informações, a quantidade de informações repetidas será imensa, o que gera custos para armazenamento de dados. Além disso, com o passar do tempo e as operações, poderão ocorrer erros de exatidão das informações. Bastará um funcionário do Setor de Vendas atualizar errado o nome de uma passageira e isso fará com que as informações dessa passageira fiquem diferentes das que estão nos arquivos dos outros setores.

Assim, existe um problema grave de consistência das informações e, também, de redundância (duplicidade) das informações. Somando-se a isso, a existência de diversas cópias de arquivos produz uma diminuição na segurança do sistema como um todo, pois pessoas mal-intencionadas poderão acessar esses arquivos, sem controle algum, e produzir danos na reputação da empresa.

Para sanar essas dificuldades todas, usam-se sistemas de Bancos de Dados.

Um sistema de Banco de Dados funciona em um computador ligado a uma rede de computadores. Ele unifica as várias cópias dos arquivos que antes estariam dispersas pelos vários setores da empresa. Haverá um único conjunto de arquivos com as informações necessárias, centralizado no computador que armazena o Banco de Dados.

Todos os setores poderão acessar esse computador (o Servidor de Bancos de Dados) através da rede de computadores. Um sistema de rede de computadores possui módulos de segurança que busca evitar acessos não autorizados.

Já o Servidor de Bancos de Dados possuirá módulos que controlarão a segurança, permitindo o acesso que cada setor pode ter aos dados, apenas para usuários autorizados; módulos para atualização dos dados de forma que fiquem atualizados nesse servidor e não haja diferenças nas informações; com ele se evita a redundância, pois ao invés de cópias dos arquivos espalhadas pela empresa, teremos uma única versão desses arquivos, centralizada no servidor. Dessa maneira, os dados ficam integrados num único local da rede da empresa.



Com isso, as empresas, mesmo as pequenas, podem ter maior controle sobre suas informações e operar com maior eficiência, segurança e qualidade.

Os Bancos de Dados são usados, atualmente, na maioria das aplicações computacionais mais difundidas na sociedade. Mesmo quando você acessa um site no celular ou num computador, na maioria das vezes existe um sistema de bancos de dados utilizado para buscar as informações que são apresentadas.

2. Definições Iniciais

Dados

São fatos isolados em uma forma primitiva, primária, que podem ser registrados em algum meio, sem estarem associados a algum uso específico. Por exemplo: RG, CPF, Nome, Data, Código, Comprimento, Cor.

Por si só, dados não tem muita significância. Por exemplo, uma data qualquer, por si só, não significa muita coisa. O mesmo pode ser pensado a respeito de um nome de uma pessoa qualquer.

Mas, eventualmente, um dado isolado pode ser associado a outros dados isolados e dar origem ao que chamamos de informação.

Bancos de Dados são usados para armazenar dados de diferentes assuntos e naturezas.

Informação

São os fatos organizados de tal maneira que passam a produzir um significado. Em geral, a informação está associada a algum uso específico, dentro de um contexto. Por exemplo: Lista de clientes com seus nomes, CPF e datas de nascimento, ordenados alfabeticamente pelos nomes dos clientes.

Bancos de dados são usados para organizar os dados isolados de maneira que, através dessa organização, seja possível trata-los de forma integrada e extrair informações a partir dos dados.

Dados

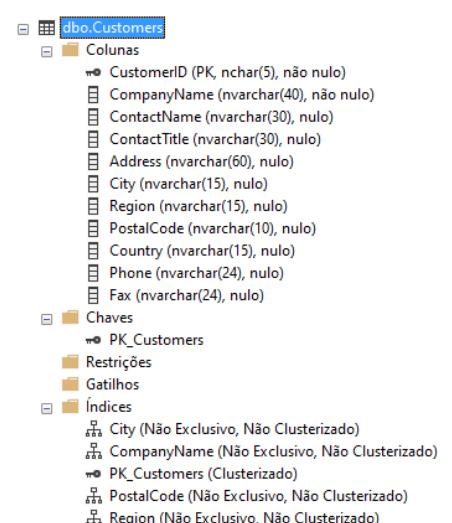
Informação



Metadados

Há também um conceito importante, os metadados, que são “dados sobre os dados”. Em outras palavras, para cada dado isolado de mesmo tipo armazenado, existe uma descrição sobre esses dados, ou seja, dados organizados (informações) que dizem respeito aos próprios dados, indicando como os dados serão representados no banco de dados, como identificar cada dado, garantindo consistência (correção dos dados) e persistência (seu armazenamento em uma unidade de armazenamento, para não serem perdidos futuramente).

Os metadados são mantidos em um local especial do Banco de Dados, chamado Dicionário de Dados. Esse dicionário de dados descreve a estrutura física de cada dado (tipo, tamanho, se pode ser ou não armazenado sem conteúdo) e outras descrições importantes para o funcionamento do banco de dados.



Banco de Dados

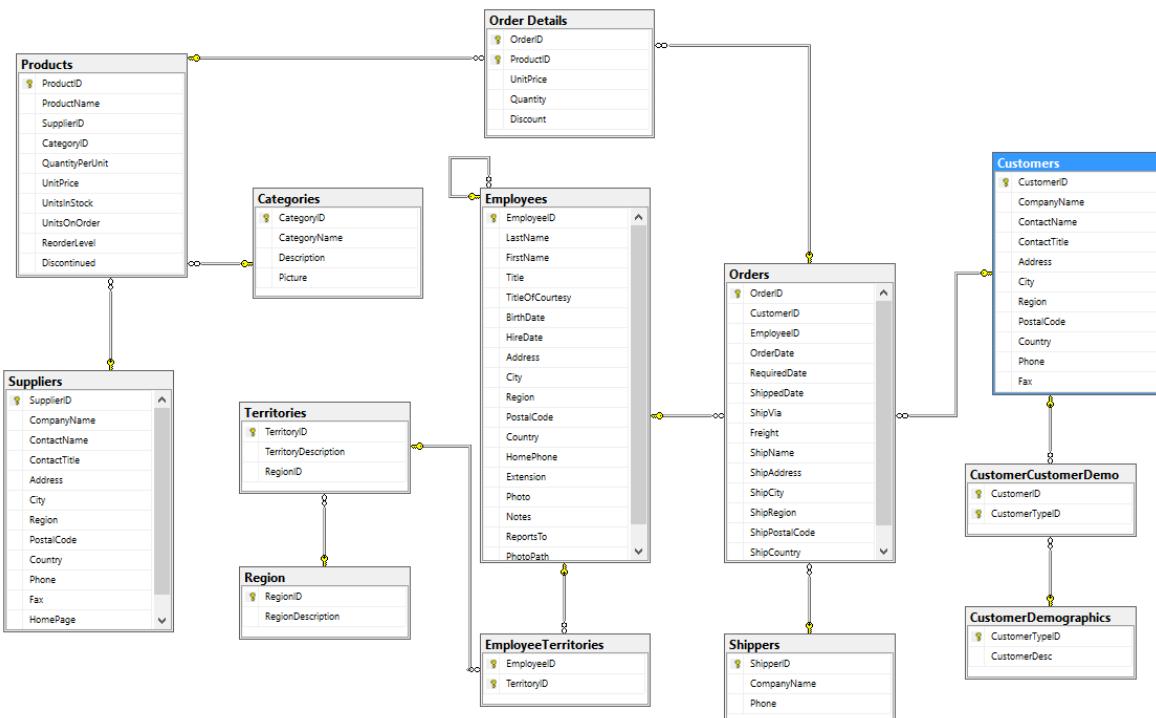
É uma coleção organizada de dados. Esses dados são organizados de modo a modelar aspectos do mundo real, para que seja possível efetuar processamento que gere informações relevantes para os usuários a partir desses dados.

Há vários conjuntos de dados, que são organizados (dispostos, arranjados, enfileirados, etc) de modo a facilitar a busca dos mesmos; eles não ficam armazenados no banco de dados de qualquer maneira, aleatoriamente distribuídos.

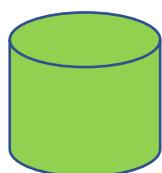
Por exemplo, se quisermos modelar um banco de dados com os dados dos alunos desta classe, vamos ter que modelar no computador, usando um software, a descrição desses dados (identificação de cada um, tamanho máximo que ocupará na memória do computador, natureza desse dado). Usaremos dados e conceitos que existem no mundo real, como o nome do aluno, seu e-mail, data de nascimento, curso em que está matriculado, telefone, dentre outros dados referentes a cada aluno específico.

Com esses dados e a organização que o sistema de banco de dados proporciona, podemos guardar informações sobre os alunos, efetuar pesquisas buscando algum aluno específico ou um grupo de alunos cujas informações combinem com algum critério de seleção, incluir novos alunos, alterar dados de alunos existentes ou mesmo excluir um ou mais alunos.

Essas operações são proporcionadas pelo Sistema de Banco de Dados, que é composto por diversos objetos, como tabelas, relacionamentos, índices, esquemas, visões, consultas, relatórios, procedimentos e funções armazenados, disparos, dentre outros. Também controla contas de usuários que podem acessar (ou não) esses objetos, além de outros aspectos da utilização de um sistema de Banco de Dados.



Este é o símbolo extraoficial comumente usado em diagramas para representar um Banco de Dados.



SGBD – Sistema de Gerenciamento de Banco de Dados

Um SGBD é uma coleção de softwares que permite aos usuários criarem e manterem um ou mais bancos de dados.

Como vimos acima, um Banco de Dados é composto por diversos objetos, como se fossem vários arquivos isolados. Cada arquivo é responsável por um aspecto do armazenamento dos dados. O SGBD permite tratar de todos esses arquivos de maneira integrada, coerente e segura.

Portanto, um SGBD integra diversas tarefas sob um único software. Tais tarefas são, por exemplo: definição de um banco de dados, construção de objetos de um banco de dados, manutenção e pesquisa dos dados, compartilhamento do banco de dados entre aplicações e usuários.

Também realizam tarefas de proteção, consistência e validação dos dados.

Há vários sistemas de gerenciamento de bancos de dados no mundo atual. Dentre eles, podemos citar:



Exemplo de SGBDR –Tela do Gerenciador de Banco de Dados Relacional Sql Server

The screenshot shows the Microsoft SQL Server Management Studio (SSMS) interface. On the left, the Object Explorer tree view displays the database structure for the 'northwind' database, including tables like Employees, Customers, and Products. On the right, a query window titled 'instnwnd.sql - ERIDANI.northwind (ERIDANI\chico (58))' contains the following SQL code:

```

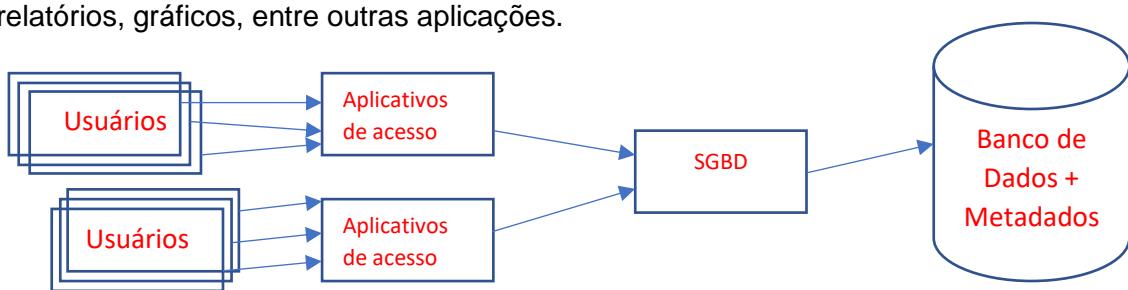
GO
CREATE TABLE "Employees" (
    "EmployeeID" "int" IDENTITY (1, 1) NOT NULL ,
    "LastName" nvarchar (20) NOT NULL ,
    "FirstName" nvarchar (10) NOT NULL ,
    "Title" nvarchar (30) NULL ,
    "TitleOfCourtesy" nvarchar (25) NULL ,
    "BirthDate" datetime NULL ,
    "HireDate" datetime NULL ,
    "Address" nvarchar (60) NULL ,
    "City" nvarchar (15) NULL ,
    "Region" nvarchar (15) NULL ,
    "PostalCode" nvarchar (10) NULL ,
    "Country" nvarchar (15) NULL ,
    "HomePhone" nvarchar (24) NULL ,
    "Extension" nvarchar (4) NULL ,
    "Photo" image NULL ,
    "Notes" ntext NULL ,
    "ReportsTo" int NULL ,
    "PhotoPath" nvarchar (255) NULL ,
    CONSTRAINT "PK_Employees" PRIMARY KEY CLUSTERED
    (
        "EmployeeID"
    ),
    CONSTRAINT "FK_Employees_Employees" FOREIGN KEY
    (
        "ReportsTo"
    ) REFERENCES "dbo"."Employees" (
        "EmployeeID"
    ),
    CONSTRAINT "CK_Birthdate" CHECK (BirthDate < getdate())
)
GO
CREATE INDEX "LastName" ON "dbo"."Employees"("LastName")
GO
CREATE INDEX "PostalCode" ON "dbo"."Employees"("PostalCode")
GO

CREATE TABLE "Categories" (
    "CategoryID" "int" IDENTITY (1, 1) NOT NULL ,
    "CategoryName" nvarchar (15) NOT NULL ,
    "Description" ntext NULL ,
    "Picture" image NULL ,
    CONSTRAINT "PK_Categories" PRIMARY KEY CLUSTERED
    (
        "CategoryID"
    )
)

```

Esse programa (Sql Server Management Studio) é um dos vários que fazem parte do pacote do Sql Server. Com ele podemos visualizar os bancos de dados criados, os objetos internos de cada banco de dados e executar códigos na linguagem SQL que permite manipular os objetos e os dados armazenados.

Programas como esse podem ser desenvolvidos para acessar um banco de dados específico. Esse acesso pode ser feito por aplicações desktop, mobile e/ou web, dentre outros tipos. Esses programas são executados pelos usuários, a fim de cadastrar e recuperar informações, gerar relatórios, gráficos, entre outras aplicações.



Observe que o banco de dados é um elemento e o SGBD é outro elemento. A interconexão de todos esses elementos permite o uso de sistemas de bancos de dados.

Os usuários de um sistema de banco de dados podem ser de diversos tipos:

- Usuário final: executa o aplicativo de acesso para processar informações
- Projetista / Desenvolvedor: modela e projeta o banco de dados para que possa ser administrado e acessado
- Administrador (DBA): gerencia o banco de dados, faz sua manutenção, cópias de segurança

É comum que um usuário possa acumular duas ou mais das funções acima.

Características e Funcionalidades

Um banco de dados moderno fornece algumas funcionalidades que facilitam seu uso e o tornam bastante útil, como:

- Controle de Redundância – evita o armazenamento de informações repetidas sem necessidade (duplicidade de dados)
- Múltiplas visões dos dados – diferentes usuários, com níveis de acesso ou necessidades distintos, podem ter visualizações distintas do mesmo banco de dados
- Controle de concorrência – evita conflitos de acesso, alteração e exclusão de dados que estão sendo acessados simultaneamente por diferentes usuários
- Backup e Restauração – operações de segurança dos arquivos pertencentes ao banco de dados, para que sempre haja uma cópia de segurança.
- Autenticação e Autorização de acesso – somente pessoas autorizadas podem acessar os recursos do banco de dados
- Restrições de Integridade – aplicar regras sobre os dados para que estes fiquem corretos. Por exemplo, um aluno não pode ser cadastrado no banco de dados se o curso em que se matriculou não estiver previamente cadastrado. Sem isso, as informações desse aluno perdem a sua integridade, confiabilidade.

Essas funcionalidades são configuradas em um banco de dados e, a partir daí, o sistema de banco de dados passa a executá-las automaticamente, sem que seja necessário que o desenvolvedor da aplicação tenha que se preocupar em implantá-las no seu software de aplicação, de forma que estes ficam mais simples, pois boa parte da infraestrutura de utilização do banco de dados é controlada pelo próprio sistema de gerenciamento do banco.

Aplicações dos Bancos de Dados

Há uma enormidade de aplicações computacionais dos Bancos de Dados. Por exemplo:

- Sistemas bancários
- Reservas em hotéis
- Passagens aéreas, de ônibus, de trem, navio
- Controle de estoque em empresas varejistas
- Bibliotecas
- Comércio eletrônico, lojas virtuais
- Receita Federal
- Correios
- Redes sociais
- Youtube

Cada uma dessas (e de inúmeras outras) aplicações usam sistemas de Bancos de Dados para armazenar e organizar dados referentes ao negócio/empreendimento que operam, de forma que possam recuperá-los como informações úteis à gestão do negócio.

Tipos de Bancos de Dados

No decorrer da história da Computação, diversas formas de implantação de sistemas de estruturação de bancos de dados foram desenvolvidas, sempre buscando otimizar a manutenção e recuperação das informações armazenadas, de acordo com os recursos computacionais disponíveis em cada época.

Assim, temos a seguinte relação de tipos de bancos de dados, a maioria deles já obsoleta:

Manual

Antigamente os dados eram escritos em fichas de papel, guardadas em uma caixa ou gaveta, organizadas por alguma maneira para acesso menos lento. Havia muita redundância de informações, integridade dos dados era falha, segurança era baixa.

Como o preenchimento era manual, sem possibilidade de verificação imediata se os dados estavam corretos, a possibilidade de erros era alta.



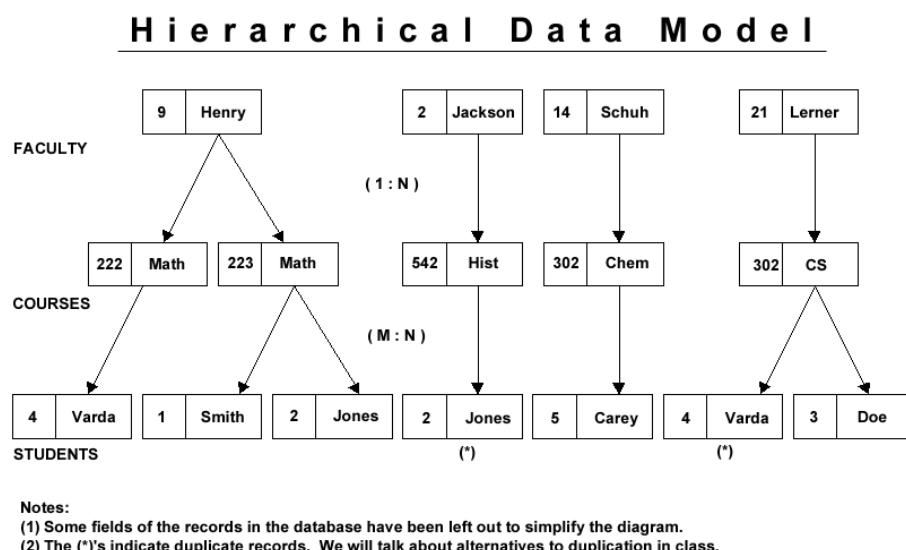
Banco de Dados Hierárquico

Nesse modelo, os dados são organizados de forma hierárquica, em que um dado serve como origem de outros dados diretamente ligados a ele, com conjuntos de dados agrupados (registros) e interconectados por meio de ligações.

Uma ligação entre dados representa uma relação entre dois tipos de registros: pai e filho, antecessor e descendente, superior e subordinado.

Um organograma é uma representação hierárquica.

O acesso aos dados sempre se dá de forma unidirecional, ou seja, a partir do dado pai se chega ao dado filho. Um dado filho pode também ser pai de outros dados subordinados a ele.

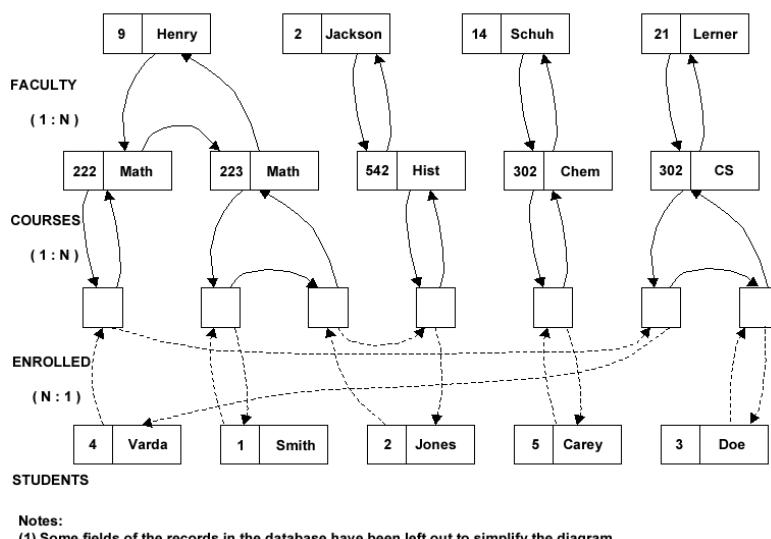


Banco de Dados em Rede

Nesse modelo, os dados são organizados em tipos e ligações entre dois registros, sem que haja restrições hierárquicas.

O formato dos agrupamentos de dados é baseado em um conceito computacional chamado grafo direcionado.

Network Data Model



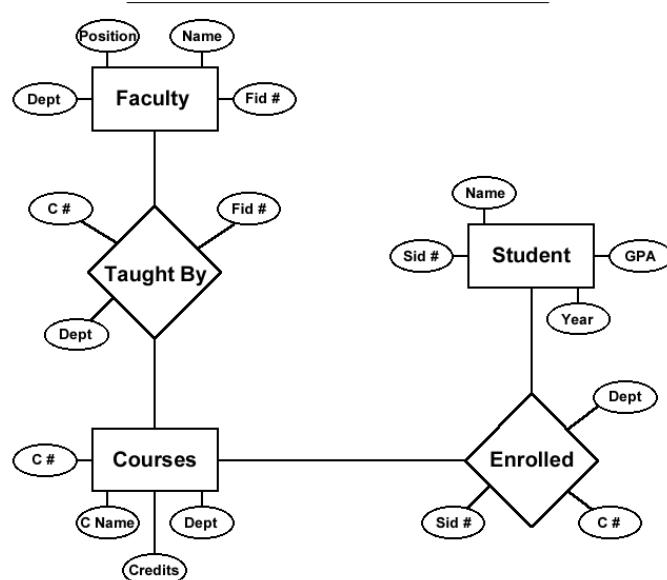
Banco de Dados Relacional

Nesse modelo, os dados são agrupados em entidades, conforme cada assunto, e registrados como atributos dessas entidades. Um atributo é uma característica que descreve uma entidade.

Por exemplo, se tivermos uma entidade Aluno, nela colocaremos os atributos que descrevem um aluno, como nome, data de nascimento, RA, data do vestibular, data de matrícula, dentre outros. Assim, a entidade Aluno descreverá as características de um aluno, a partir de todos os atributos que ela armazena.

As entidades são ligadas por meio de relacionamentos, que indicam como diferentes entidades podem ser tratadas em conjunto para podermos obter informações mais abrangentes. Por exemplo, podemos ter uma entidade de Disciplinas Oferecidas na escola, e relacionarmos as disciplinas lecionadas na escola com os alunos matriculados em cada uma delas.

E - R Data Model



Relational Data Model

Student Relation			
Sid #	Name	Year	GPA
1	Smith	3	3.0
2	Jones	2	3.5
3	Doe	1	1.2
4	Varda	4	4.0
5	Carey	4	0.5

Faculty Relation			
Fid #	Name	Position	Dept
9	Henry	Prof.	Math
2	Jackson	Assist. Prof	Hist
14	Schuh	Assoc. Prof	Chem
21	Lerner	Assist. Prof	CS

Course Relation			
Course #	Course Name	Cr	Dept
223	Calculus	5	Math
302	Intro Prog	3	CS
302	Organic Chem	3	Chem
542	Asian Hist	2	Hist
222	Calculus	5	Math

Taught-By Relation		
C #	Fid #	Dept
223	9	Math
222	9	Math
302	21	CS
302	14	Chem
542	2	Hist

Enrolled Relation		
Sid #	C #	Dept
1	223	Math
4	222	Math
4	302	CS
3	302	CS
5	302	Chem
2	542	Hist
2	223	Math

Banco de Dados Orientado a Objetos

Fonte: https://pt.wikipedia.org/wiki/Banco_de_dados_orientado_a_objetos

Um **banco de dados orientado a objetos** é um [banco de dados](#) em que cada informação é armazenada na forma de [objetos](#), ou seja, utiliza a [estrutura de dados](#) denominada [orientação a objetos](#), a qual permeia as [linguagens](#) mais modernas. Começou a ser comercialmente viável em 1980. O [gerenciador do banco de dados](#) para um orientado a objeto é referenciado por vários como [ODBMS](#) ou [OODBMS](#).

Existem dois fatores principais que levam à adoção da tecnologia de banco de dados orientados a objetos. A primeira, é que, em um [banco de dados relacional](#), se torna difícil de manipular com dados complexos (esta dificuldade se dá pois o modelo relacional se baseia menos no senso comum relativo ao modelo de dados necessário ao projeto e mais nas contingências práticas do armazenamento eletrônico). O segundo fator é que os dados são geralmente manipulados pela [aplicação](#) escrita usando linguagens de programação [orientada a objetos](#), como [C++](#), [C#](#), [Java](#), [Python](#) ou [Delphi](#) (Object Pascal), e o código precisa ser traduzido entre a representação do dado e as tuplas da tabela relacional, o que além de ser uma operação tediosa de ser escrita, consome tempo. Esta perda entre os modelos usados para representar a informação na aplicação e no banco de dados é também chamada de "perda por resistência".

Num banco de dados orientado a objetos puro, os dados são armazenados como objetos onde só podem ser manipulados pelos métodos definidos pela [classe](#) de que estes objetos pertencem. Os objetos são organizados numa [hierarquia](#) de tipos e subtipos que recebem as características de seus supertipos. Os objetos podem conter referências para outros objetos, e as aplicações podem, assim, acessar os dados requeridos usando um estilo de navegação de programação.

Um dos objetivos de um SGBDO (Sistema de Gerenciamento de Banco de Dados de Objeto) é manter uma correspondência direta entre objetos do mundo real e do banco de dados, de modo que objetos não percam sua integridade e identidade e possam facilmente ser identificados e operados. Assim, o SGBDO oferece uma identidade única e imutável a cada objeto armazenado no banco de dados chamado [OID](#)(Identificador de objeto). O valor de um [OID](#) não é visível ao usuário externo, somente ao sistema para poder gerenciar a referência dos objetos.^[3]

A maioria dos bancos de dados também oferece algum tipo de linguagem de consulta, permitindo que os objetos sejam localizados por uma programação declarativa mais próxima. Isto é, na área das linguagens de consulta orientada a objetos. A integração da consulta com a interface de navegação faz a grande diferença entre os produtos que são encontrados. Uma tentativa de padronização foi feita pela ODMG (*Object Data Management Group*) com a OQL (*Object Query Language*).

O acesso aos dados pode ser rápido porque as junções geralmente não são necessárias (como numa implementação tabular de uma [base de dados relacional](#)), isto é, porque um objeto pode ser obtido diretamente sem busca, seguindo os ponteiros.

Outra área de variação entre os produtos é o modo que este esquema do banco de dados é definido. Uma característica geral, entretanto, é que a linguagem de programação e o esquema do banco de dados usam o mesmo modo de definição de tipos.

Aplicações [multimídias](#) são facilitadas porque os métodos de classe associados com os dados são responsáveis pela correta reprodução.

Muitos bancos de dados orientados a objetos oferecem suporte a [versões](#). Um objeto pode ser visto de todas as várias versões. Ainda, versões de objetos podem ser tratadas como objetos na versão correta. Alguns bancos de dados orientados a objetos ainda proveem um suporte sistemático a [triggers](#) e [constraints](#) que são as bases dos bancos ativos.

Atualmente há ferramentas para compatibilizar bancos de dados relacionais com linguagens de programação orientadas a objetos, como o Entity Framework da Microsoft, usada para realizar um mapeamento de Modelo Relacional para Modelo de Objetos, de forma que um programa possa acessar um banco de dados relacional usando recursos próprios da Programação Orientada a Objetos.

Evolução de Modelos de Banco de Dados

Evolução dos Bancos de Dados

(Khoshafian 1995)

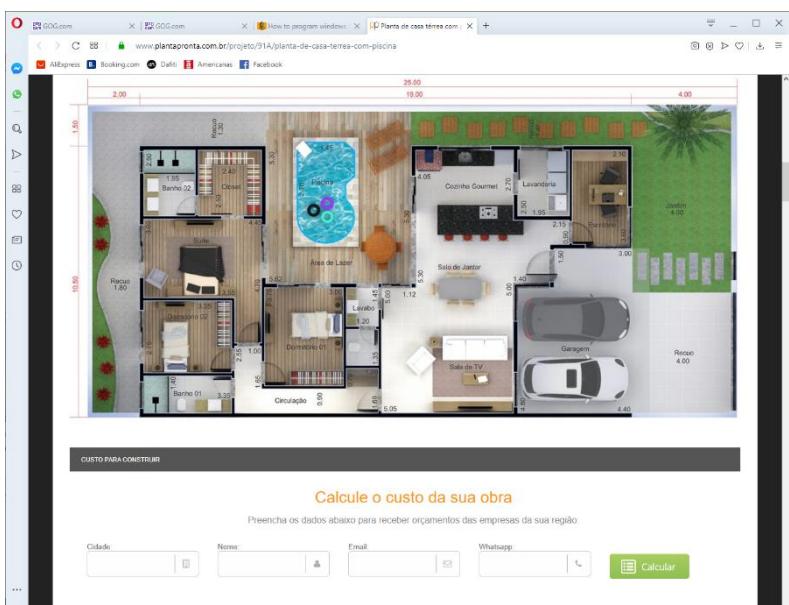


3. Modelagem de Dados

3.1. Modelo

Um modelo é uma estrutura abstrata que ajuda a pensar e organizar os conceitos que estão na mente de um projetista, bem como comunicá-los para outras pessoas.

Por exemplo, a planta de uma casa: é uma estrutura abstrata porque não é o objeto concreto (uma casa) já construído, e pode ser usada para realizar tarefas tais como descrever, analisar, especificar e comunicar ideias.



Fonte: www.plantapronta.com.br

Observe, também, que esse mesmo modelo está sendo usado em uma página da Internet que permite calcular o custo de uma obra. Portanto, um modelo pode ser muito útil para uso em aplicações computacionais, como um guia, uma diretriz para realização de diversas tarefas sobre informações do processo associado a essa aplicação.

No caso específico de Bancos de Dados, um modelo servirá para pensarmos em todos os dados de um processo e como se relacionam, como se correspondem e agem entre si.

O modelo deverá ter detalhes suficientes para que um desenvolvedor consiga construir o banco de dados de acordo com as necessidades (requisitos) do processo.

O modelo é uma representação simples, normalmente textual e/ou gráfica, de estrutura de dados reais mais complexas. Sua função é auxiliar na compreensão das complexidades do ambiente real de negócios, representando estruturas de dados e suas características, relações, restrições, transformações e outros elementos que tenham finalidade de dar suporte ao problema específico de um negócio.

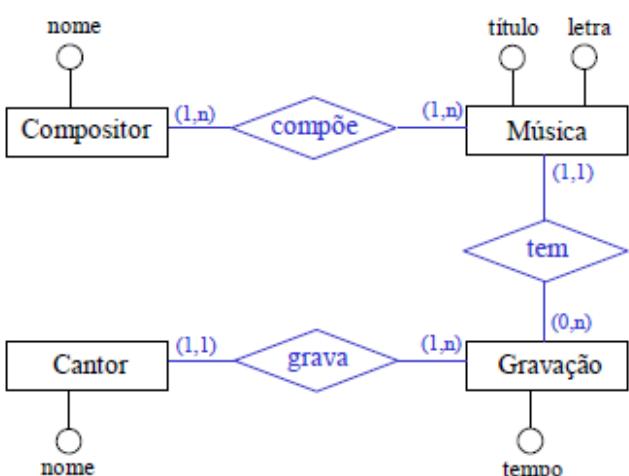
Ao lado temos um modelo de dados bastante simples, que explicita as relações entre entidades Compositor, Música, Cantor e Gravação:

Observe a figura ao lado. É uma planta de uma casa, e se trata de um modelo da casa.

O modelo poderia ser também uma maquete física, que demandaria maior trabalho para realizar e, de certa maneira, dificultaria a visualização da casa internamente.

De maneira semelhante, a planta baixa da casa não permite visualizar como seriam as paredes, janelas, etc. Esses elementos seriam melhor visualizados com a maquete.

Dessa forma, vários diferentes tipos de modelos podem ser usados para comunicar ideias referentes às mesmas **entidades**.



Fonte: <https://www.diegomacedo.com.br/modelagem-conceitual-logica-e-física-de-dados/>

3.2. Processo de Modelagem de Dados

Modelagem de Dados é o processo de criação de um Modelo de Dados a ser usado em um sistema de informação, com a aplicação de técnicas específicas de modelagem.

Essas técnicas envolvem a definição e análise de requisitos de dados necessários para suportar as atividades de uma organização. Essas atividades são chamadas, para efeito de simplicidade, de “negócios”. Após essa análise de requisitos, o desenvolvedor irá criar modelos que permitam visualizar os dados do processo sob análise e como se relacionam entre si, para que seja possível automatizar os negócios de forma eficiente e segura.

Portanto, a modelagem de dados é um processo no qual você planeja como será a sua base de dados (ou banco de dados), para refletir as características do negócio.

A modelagem de dados tem por objetivos:

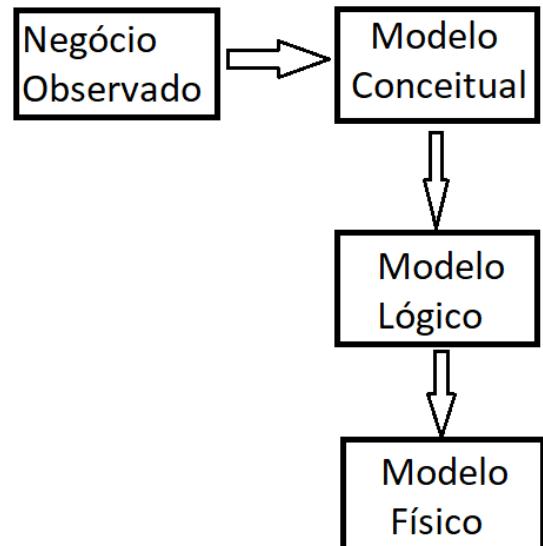
- Representar os conceitos do ambiente de negócios observado
- Documentar e normalizar os dados no modelo de dados
- Fornecer processos de validação
- Observar fatos no ambiente de negócios e identificar relacionamentos entre os objetos do ambiente de negócios.
- Representar as regras de negócio que regulam os processos no ambiente de negócios

Para se obter uma boa modelagem dos dados, através de um projeto de banco de dados, faz-se necessário o conhecimento das regras de negócio, ou seja, das premissas e restrições inerentes ao negócio. Para se conseguir isso é feita uma análise de todas as informações que fazem parte do negócio a fim de se determinar de que forma estas informações serão tratadas.

Para haver um perfeito entendimento e compreensão das necessidades de informação do negócio que está sendo atendido pelo projeto do banco de dados, é necessário um bom levantamento de requisitos. Essa atividade que é feita pelo analista de requisitos que atua junto ao analista de negócios para descobrir os requisitos de informação que devem ser atendidos no projeto do banco de dados.

O modelo de dados fornecerá uma estrutura para os dados usados em um sistema de informações, com definições e formatos específicos de cada dado.

Muitas vezes, na pressa de implementar um Banco de dados, a fase de modelagem é deixada de lado. Isso acarreta vários problemas futuros, como falta de informações, definições errôneas de tipos de dados, informações redundantes, dentre outros problemas.



Arquitetura de 3 níveis

3.2.1. Visão Geral do Modelo Relacional

Devido às deficiências e dificuldades no uso de sistemas de bancos de dados baseados nos modelos Hierárquico e de Rede, em junho de 1970, Edgar Frank Codd (1923-2003) propôs, no artigo intitulado “A Relational Model of Data for Large Shared Data Banks”, um novo modelo, o Relacional. Nesse artigo o autor delineia regras para a modelagem relacional de dados, visando a criação de sistemas de informações apoiados por bancos de dados.

O artigo descreve as operações matemáticas necessárias para a criação e uso de bancos de dados relacionais. Para uma visão geral que Codd tinha sobre como deveria ser um banco de dados relacional, consulte [aqui](#).

Relações e tabelas

No Modelo Relacional, os dados que fazem parte do problema a ser resolvido são agrupados e organizados em coleções de dados, chamadas de Relações ou Tabelas.

Os dados que possuem alguma origem ou aplicação comum são agrupados em linhas e colunas, como uma tabela de dados, de forma semelhante à que vemos na figura ao lado. Essa ideia é baseada na teoria dos conjuntos da Matemática.

Indivíduo	Tabela B	
	X: Massa corporal (kg)	Y: Consumo anual de água (L)
1	90	850
2	120	400
3	60	300
4	40	550
5	82	490
6	90	350

Composição do modelo relacional

- coleções de objetos ou relações que armazenam os dados (entidades, tabelas)
- operadores que agem nas relações, produzindo outras relações
- integridade de dados, para precisão e consistência

O resultado da modelagem relacional de um negócio poderá ser usado para criar fisicamente um banco de dados, que será composto por tabelas (semelhantes a arquivos) para armazenar os dados, além de informações sobre como as diferentes tabelas se relacionam.

Por exemplo, se desejamos criar um banco de dados para uma loja, teremos de criar tabelas para relacionar os clientes da loja (dados pessoais, dados de compras, dados de crédito, dentre outros), os produtos vendidos (descrição, preço, quantidade disponível em estoque), vendedores (nome, relação de produtos vendidos a quais clientes, data de cada venda), fornecedores (contato, endereço, relação de produtos fornecidos, preço de cada produto). Muito possivelmente esses dados serão armazenados em diferentes tabelas, para aumentar a segurança, flexibilidade e consistência no acesso aos dados. Esse banco de dados será criado a partir do modelo relacional, fruto do processo de modelagem de dados que vamos aprender.

3.3. Fases da modelagem

A Modelagem de Dados envolve várias fases, como Análise de Requisitos, Modelo Conceitual, Modelo Lógico, Análise de Dependências, Normalização, Modelo Físico. A partir daí, o Banco de Dados poderá ser usado para automatizar processos através de programas externos a ele e, também de programas internos, através de Procedimentos e Funções Armazenados, Triggers e Visões.

3.3.1. Análise de Requisitos e Modelo Conceitual

A análise de requisitos é uma fase extremamente importante para o sucesso do projeto de um Banco de Dados.

Elas se iniciam com reuniões com os clientes e/ou usuários do sistema, para coleta de informações sobre o negócio a ser suportado pelo banco de dados. Nessas reuniões são identificadas entidades que fazem parte do negócio, as associações entre elas e os atributos de cada entidade. Essas informações são registradas, analisadas mais profundamente e documentadas.

Conforme as reuniões vão ocorrendo, a documentação vai sendo melhorada (refinada) com a agregação de mais informações e o conhecimento aprimorado das regras de negócio. Um outro aspecto da Análise de Requisitos é descartar dados que não sejam relevantes para o negócio. Por exemplo, num sistema de controle escolar não seria útil ter um campo para a idade dos alunos, pois

essa varia a cada dia. Mais interessante seria ter um campo com a data de nascimento, a partir da qual a idade real do aluno pode ser calculada facilmente.

Se a Análise de Requisitos não for bem feita, futuramente o banco de dados modelado a partir dela apresentará falhas.

Portanto, os objetivos principais da Análise de Requisitos são:

- Descobrir quais são os dados básicos da empresa e, a partir deles, delinear os requisitos de dados;
- Descrever a informação sobre esses dados e agrupá-los em entidades descritas por grupos de dados;
- Descrever os relacionamentos entre os agrupamentos de dados (as entidades)
- Determinar as operações que devem ser executadas no banco de dados para atingir os objetos dos processos da empresa;
- Definir como essas operações utilizam e afetam os dados;
- Definir quaisquer restrições de desempenho, integridade, segurança e de administração que tenham de ser observadas durante o uso do banco de dados e, de preferência, gerenciadas por ele;
- Especificar quaisquer restrições/limitações de projeto e de implementação, tais como tecnologias, hardware e software, linguagens de programação, políticas de acesso e de uso dos dados, padrões ou interfaces para que o banco de dados se comunique com agentes externos
- Documentar completamente todos os itens acima em uma especificação de requisitos detalhada. Pode-se também usar um sistema de Dicionário de Dados para definir cada dado levantado, seu significado, formato e uso

A partir das informações obtidas na Análise de Requisitos, será criado o modelo Conceitual do Banco de Dados. Nele determinamos quais informações precisam ser armazenados no banco de dados. Neste nível, o projeto ainda é independente do Sistema de Gerenciamento de Banco de Dados em que o sistema será implantado fisicamente.

Embora detalhes de implementação ainda não apareçam, já é possível descrever os tipos de cada dado requerido, os relacionamentos entre as entidades e as regras de consistência.

Exercícios

Identifique os Dados e as Entidades envolvidas nas situações abaixo:

a) Um(a) estudante estuda em uma escola técnica, na qual realiza um ou mais cursos. É identificado por um Registro Acadêmico (RA), mas é importante conhecer seu nome, endereço, data de nascimento, gênero, bem como os cursos em que está matriculado. Os cursos oferecidos pela escola são todos identificados por um código, mas também temos de saber os nomes dos cursos, turnos de oferecimento, quantos períodos letivos formam cada curso. Os estudantes também realizam disciplinas dentro de cada período letivo e, para identificar cada disciplina, temos seus códigos, nomes, cargas horárias. Além disso, uma disciplina é lecionada por um ou mais professores, cada um deles sendo responsável por uma turma daquela disciplina.

b) Uma **floricultura** deseja informatizar suas operações. Inicialmente, deseja manter um cadastro de todos os seus clientes, mantendo informações como: RG, nome, telefone e endereço. Deseja também manter um cadastro contendo informações sobre os produtos que vende, tais como: nome do produto, tipo (flor, vaso, planta, etc.), preço e quantidade em estoque. Quando um cliente faz uma compra, a mesma é armazenada, mantendo informação sobre o cliente que fez a compra, a data da compra, o valor total e os produtos comprados.

c) Uma **biblioteca** deseja manter informações sobre seus livros. Inicialmente, quer armazenar para os livros as seguintes características: ISBN, título, ano editora e autores deste livro. Para os autores, deseja manter: nome e nacionalidade. Cabe salientar que um autor pode ter vários livros, assim

como um livro pode ser escrito por vários autores. Cada livro da biblioteca pertence a uma categoria. A biblioteca deseja manter um cadastro de todas as categorias existentes, com informações como: código da categoria e descrição. Uma categoria pode ter vários livros associados a ela.

d) Uma firma vende **produtos de limpeza**, e deseja melhor controlar os produtos que vende, seus clientes e os pedidos. Cada produto é caracterizado por um código, nome do produto, categoria (ex. detergente, sabão em pó, sabonete, etc.), e seu preço. A categoria é uma classificação criada pela própria firma. A firma possui informações sobre todos seus clientes. Cada cliente é identificado por um código, nome, endereço, telefone, status ("bom", "médio", "ruim"), e o seu limite de crédito. Guarda-se igualmente a informação dos pedidos feitos pelos clientes. Cada pedido possui um número e guarda-se a data de elaboração do pedido. Cada pedido pode envolver de um a vários produtos, e para cada produto, indica-se a quantidade deste pedido.

Extraído e adaptado de:

<http://www.uel.br/pessoal/valerio/Lista%20de%20exercicios%20Resolvido%2001%20-%20MC%20-%206%20folhas.pdf>.

Data de acesso: 22/02/2019. Créditos ao professor: Vitor Valério de Souza Campos.

e) Uma empresa de locação de **VEÍCULOS** (número placa, modelo) os aluga para **CLIENTES** registrados (número, nome, CNH, endereço). Cada locação “presente” deve registrar o local e a data de retirada e a quilometragem do veículo, bem como o local e a data previstos para devolução. Para cada locação, o cliente deve deixar um cheque caução, sendo que o valor, o número do cheque bem como a agência e o banco devem ser registrados.

f) Uma empresa importa produtos e cada importação é registrada através de um Processo que denominaremos de “Processos de Importação” que tem (número do processo, data, local de importação). Num mesmo Processo de Importação é possível comprar (importar) produtos de diversos fornecedores estrangeiros diferentes (código de fornecedor, nome, endereço, país). Para efetuar o pagamento de cada processo de importação a empresa recebe faturas (número de fatura, valor, data de emissão, data de vencimento) dos diversos fornecedores estrangeiros que constam daquele processo. Cada processo pode ter várias faturas, porém, uma fatura deve pertencer a um único processo e obviamente ser originária de um único fornecedor.

Lista de Exercícios do Professor Israel Geraldi – Curso de Sistemas de Informação – PUCC 2019

3.3.2. Modelo Lógico

Após a definição do Modelo Conceitual, passaremos a criar o modelo Lógico e o Modelo Físico.

O Modelo Entidade-Relacionamento (MER) é criado a partir das especificações de negócio, definidas através da coleta de informações pela Análise de Requisitos, onde a narrativa dos usuários permite estabelecer as entidades envolvidas e seus detalhes.

O MER permite focar nas informações necessárias para a existência do negócio, de forma separada das atividades que são realizadas no negócio para ele funcionar.

Por exemplo, num banco de dados que tenha clientes e produtos de uma loja, focaremos nos dados necessários para representar os clientes da loja e nos dados necessários para representar os produtos que essa loja vende. Também definiremos as informações necessárias para registrar as vendas dos produtos para os clientes, ou seja, o relacionamento entre produto e cliente. No entanto, ao modelarmos essa situação, também teremos indicações de como as atividades necessárias para realizar o negócio serão feitas (dentro do que o banco de dados prevê em seu modelo).

Esse modelo permitirá conhecer as diversas interações entre as entidades que foram definidas no Modelo Conceitual, bem como dará as diretrizes para a criação de todos os objetos necessários para que o banco de dados venha a existir num servidor de Bancos de Dados e possa ser acessado.

A partir do MER criamos um diagrama que ilustra, de maneira concisa e gráfica, as diversas entidades, seus atributos e relacionamentos. Esse diagrama é o **DER** – Diagrama Entidade-Relacionamento, que nos possibilita enxergar, com detalhes e com facilidade, como será a construção do banco de dados do problema que estamos modelando.

O MER / DER é composto por vários elementos, que destacamos abaixo:

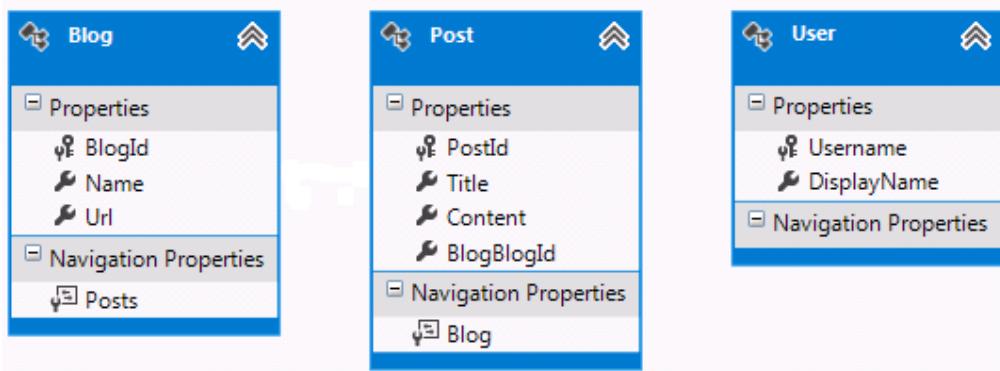
Entidade

As entidades são os principais elementos do negócio sobre os quais informações devem ser levantadas durante a modelagem. Elas normalmente representam coisas, objetos, conceitos ou situações do mundo real que tenham interesse informativo, sejam significativas e distinguíveis de outros objetos, como pessoas, lugares, eventos, produtos e conceitos abstratos, para mapear o escopo do modelo. Podem representar algo com existência física ou algo abstrato.

Em nossos diagramas, para efeito de padronização, as entidades serão representadas por retângulos e seus nomes começarão com letras maiúsculas e serão escritos no singular.



A partir delas criaremos as **tabelas** do banco de dados, nas quais as informações coletadas serão armazenadas.



Fonte: <https://docs.microsoft.com/pt-br/ef/ef6/modeling/designer/workflows/database-first>

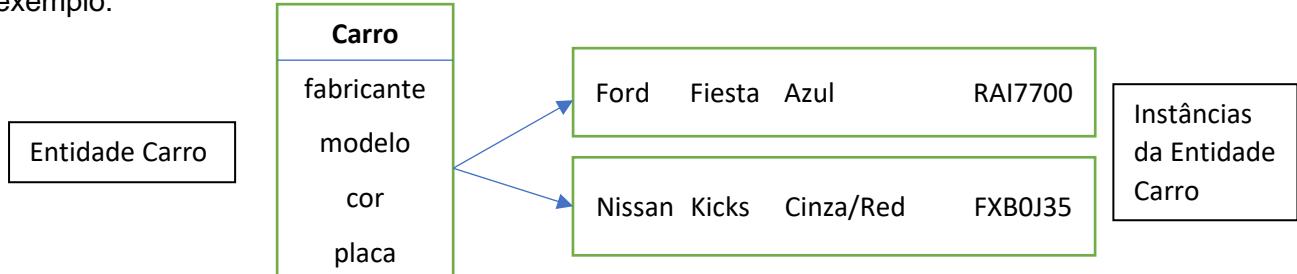
Acima podemos visualizar três entidades que já foram transformadas em **tabelas** em um sistema de gerenciamento de banco de dados. Observe que esse não é o modelo, mas sim um resultado da modelagem. As cores escolhidas para representar não são padronizadas, pode-se usar qualquer uma.

[Vídeo adicional](#)

Tupla / Linha / Registro

Cada linha de uma tabela contém dados que, trabalhados de forma integrada, representam uma determinada ocorrência de uma entidade em particular. As **linhas** de uma tabela também são chamadas de **tuplas** ou **registros**.

Os dados armazenados em uma linha específica da entidade representam uma Instância da Entidade, ou seja, um conjunto de dados que representam uma ocorrência dessa entidade. Por exemplo:



Coluna / Campo / Atributo

Em geral, as linhas de uma tabela têm uma ou mais **colunas**, também chamadas de **atributos** ou **campos**. As colunas são os dados que caracterizam uma Entidade.

Uma coluna é uma unidade de uma tabela que armazena um **tipo específico** de dado (um valor); eventualmente, essa coluna pode não armazenar nenhum valor (ou seja, está nula) mas continua fazendo parte da tabela. Por exemplo, imagine que seus dados de aluno estão armazenados em uma tabela. Neste momento, você ainda não tem nenhuma nota nas avaliações da disciplina Bancos de Dados I, mas futuramente terá. Como é bastante complicado e ineficiente colocar um campo de nota na tabela depois que o semestre letivo começou (imagine se alguém comete um erro e apaga toda a tabela com seus dados...) então já deixamos esse campo de nota previsto no modelo, mas vazio (nulo). Nem mesmo podemos deixar como um valor zero (0,0) nessa coluna, pois essa é uma nota válida, embora triste.

Na figura abaixo vemos uma possível representação da entidade **Aluno**, contendo os campos (colunas, atributos) matrícula, nome e idade, além de vários registros com dados de alunos.

TABELA/RELAÇÃO/ARQUIVO		
	MATRICULA	NOME
▶	9912333-4	Luis Fernando Matos
▶	8822266-9	Sandra Barbosa
▶	7777990-0	Júlio Vieira
▶	2234567-8	Antônio Rodrigues
▶	1400985-3	Pedro Lemos

LINHAS/
TUPLAS/
REGISTROS COLUNAS/ATRIBUTOS/CAMPOS

Fonte:<http://marcelmesmo.blogspot.com/2011/08/sistema-de-banco-de-dados-relacional.html#.Xi81k7fPyHs>

Abaixo temos um outro exemplo de tabela, com várias tuplas. Observe que alguns campos são **nulos**, ou seja, não armazenam nenhum dado.

STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
Dick Davidson	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
Barbara Benson	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25
Charles Cooper	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Katherine Ashly	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89

Fonte: <https://slideplayer.com.br/slide/1263574/>

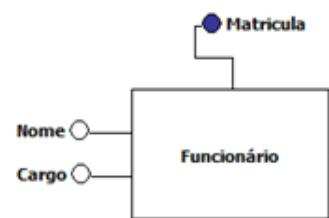
A boa prática em modelagem de dados indica que as linhas de uma tabela devem possuir um campo que identifique os dados dessa linha de forma unívoca, ou seja, sem repetições em todas as linhas da tabela. Esse campo é muito importante e é descrito como uma **chave primária**, pois permite identificar individualmente cada registro. É como se fosse o RA de um aluno, ou seu CPF, que não se repetem para nenhum outro aluno.

Nas tabelas acima, podemos imaginar os campos Matricula e NUMSEGSOCIAL como chaves primárias, já que nomes, endereços, idades podem ter repetições entre as várias linhas. Um campo que seja chave primária nunca poderá estar vazio num registro, pois é necessário para identificar a entidade representada pelo registro e vazio não identifica absolutamente nada.

Os atributos possuem um tipo e, às vezes, um tamanho máximo que ocupam no registro. Os tipos mais comuns são: inteiro, real, caractere, sequência de caracteres, data, imagem. Portanto, quando você for modelar um banco de dados, deverá pensar na natureza dos valores que serão armazenados em cada atributo.

Por exemplo, nome de pessoa e endereço deverão ser considerados como campos de tipo **sequência de caracteres**. Já o número de uma residência deverá ser considerado como um campo de tipo **inteiro**. O campo de nota de um aluno terá tipo **real**.

No Diagrama Entidade-Relacionamento, representamos os atributos como um nome ligado à entidade por meio de uma linha.

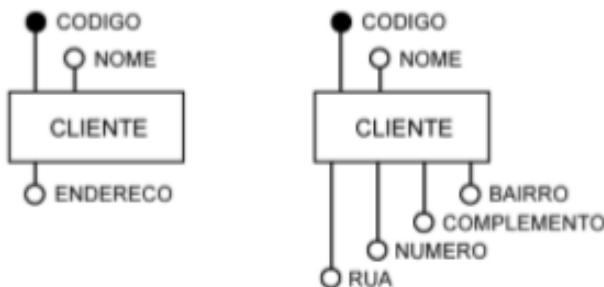


Atributos Simples

São atributos que não são divisíveis em partes. Por exemplo, RG, CPF, Nome da Empresa.

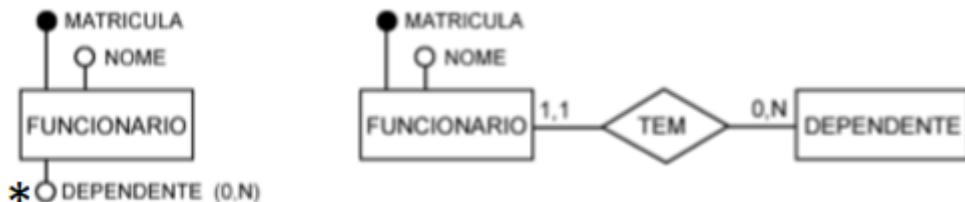
Atributos Compostos

Chamamos de atributos compostos aqueles nos quais vários itens menores. logradouro, número, complemento (exemplo: apartamento), bairro, cidade, estado, etc.



Atributos Multivvalorados

Denominamos atributo multivvalorado aquele que pode ter mais de um valor na mesma entidade. É identificado, no diagrama, por um “**”. Para atributos multivvalorados recomenda-se a divisão da entidade em duas, como na figura a seguir, onde o atributo dependente da entidade Funcionario pode ter várias ocorrências:



Essa divisão da entidade em duas faz parte de um processo de refinamento do modelo entidade-relacionamento, processo esse chamado de Normalização, que estudaremos mais à frente.

Atributo Determinante / Identificador Único

É um atributo simples ou composto que define, de forma única, as instâncias (as linhas de dados) de uma entidade. Esse atributo identifica cada instância sem que haja dubiedade, ou seja, não se repete entre as entidades.

Cada ocorrência de uma entidade deve ser identificável de forma exclusiva, sem dubiedade.

No diagrama, esse tipo de atributo pode ser representado com o seu nome sendo sublinhado.

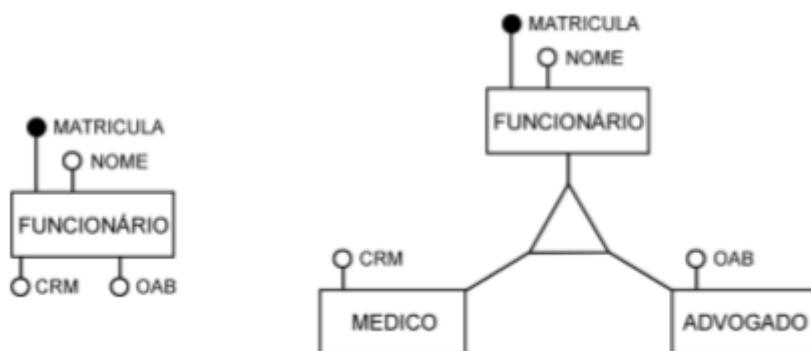
Ele poderá, também, ser usado como um atributo especial, chamado de **chave primária**, que o sistema de banco de dados usa para identificar e diferenciar cada registro de uma tabela, sem repetições entre os registros.

Como exemplos de atributos determinantes, temos CNPJ de uma empresa, CPF de uma pessoa, código de um produto, RA de um aluno, dentre outros.

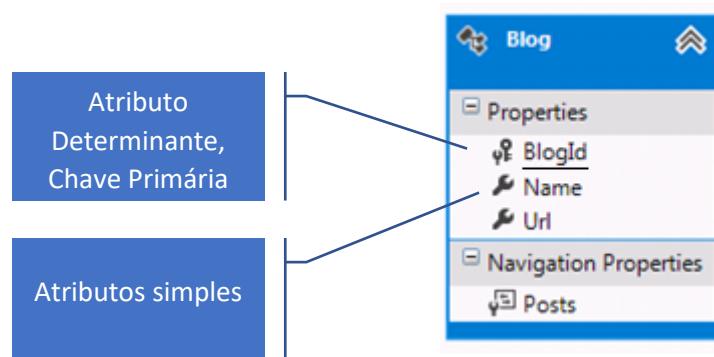
O CPF de um contribuinte brasileiro permite identificá-lo e distingui-lo de outros contribuintes, sem que se tenha dúvida sobre quem é quem, ou um código de produto nos permite identificar cada produto.

Atributos Opcionais

Atributos opcionais são aqueles que se aplicam apenas a determinadas ocorrências de uma entidade, e não a outras. Exemplo: FUNCIONÁRIO e os registros profissionais em diferentes entidades de classe: CRM, CREA, OAB, etc. Ou seja, você pode criar uma tabela separada para cada profissão relacionada.



Atributos opcionais muitas vezes indicam subconjuntos da entidade que devem ser modelados através de especialização.



Podemos também representar uma entidade e seus atributos de forma textual, como nos exemplos abaixo:

Blog (blogId, name, url)

Produto (codigo, nome, preco, quantidadeEstoque)

[Vídeo adicional](#)

Relações / Tabelas

Uma relação é uma tabela bidimensional (linhas e colunas) com características específicas baseadas na descrição de uma entidade.

As linhas da relação contêm dados sobre instâncias de uma entidade, ou seja, os registros de dados dessa entidade.

As colunas da relação representam cada atributo da entidade, ou seja, seus campos. Todos os valores dessa coluna possuem o mesmo tipo

A junção entre uma linha e uma coluna armazena um único valor, que corresponde a esse campo daquela instância da entidade.

A palavra relação denota exatamente isso: uma relação de dados dispostos em um formato tabular.

As relações darão origem a uma ou mais tabelas de dados no modelo físico, ou seja, no que será armazenado no sistema de banco de dados.

Já que é uma boa prática que cada tabela tenha um campo que identifica cada instância, não existirão duas linhas idênticas na mesma tabela.

Abaixo temos uma relação, criada a partir de uma entidade aluno.

Aluno				
ra	nome	curso	materia (*)	nascimento
20752	Aderbal Aleandro Silva	Desenvolvimento de Sistemas	(DS102, 8.5, aprovado) (DS105, 7.2, aprov) (DS101, 6.3, aprovado)	12/03/1996
19057	Alessandra Vaz	Mecanica	(TA201, 4.3, retido) (TA202, 5.7, aprovado)	25/12/1994
20708	Elaine Marques	Desenvolvimento de Sistema	(DS101, 8.0, aprovado) (DS102, 9.1, aprovado) (DS104, 5.3, aprovado)	21/06/1996
18198	Luciana Urbano	Mecânica	(ME301, 4.9, recuperac) (ME302, 5.8, aprovado) (ME303, 6.4, aprovado) (ME304, 2.9, retido)	26/10/1985
20235	Carlos Silveira	Informática	(TI101, 6.0, aprovado) (TI102, 4.5, recuper) (TI103, 7.8, aprovado)	16/05/1997
19721	Felipe Amaral	Desenvolv de Sistemas	(DS201, 4.0, retido) (DS202, 5.1, aprovado) (DS203, 6.3, aprovado)	04/06/1993

A coluna **ra** pode ser considerada um atributo determinante, a chave primária dessa tabela, pois identifica cada aluno de forma unívoca, sem dubiedade.

Note alguns **problemas** que ocorrem nas relações. Há muitos dados repetidos e palavras que podem ser escritas de formas diferentes. Outro problema grave é que uma mesma linha possui várias sublinhas no atributo **materia**, que é **multivalorado**.

Quando convertemos as relações para tabelas no modelo físico, teremos de impedir que esse tipo de coisa aconteça. Estudaremos isso no processo de criação de relacionamentos e no processo de normalização.

Vídeo adicional

Relacionamento

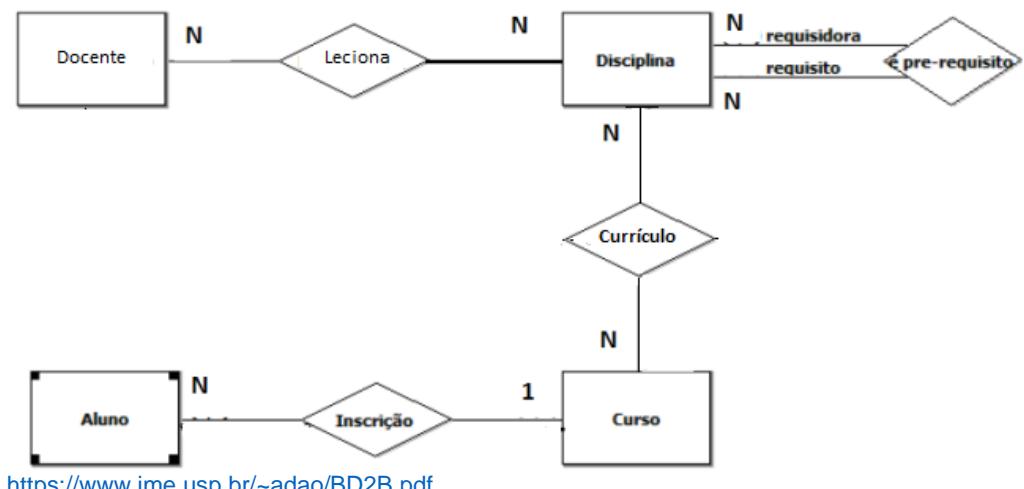
As diversas entidades que fazem parte de um negócio, em geral possuem **associações** entre si. Por exemplo, clientes podem estar associados a pedidos que estão associados a produtos e que, por sua vez, estão associados a um ou mais fornecedores.

Como outro exemplo, uma entidade Aluno está associada, em geral, a uma entidade Curso. A entidade Disciplina (matéria) está também associada à entidade Curso (associação essa chamada de Currículo). Docentes lecionam uma ou mais disciplinas. Assim, podemos dizer que alunos e cursos se relacionam, bem como disciplinas e cursos. Os alunos, além disso, se matriculam em disciplinas previstas no currículo.

Portanto, observamos que **Relacionamento** é a associação entre entidades que cria uma rede de conexões entre os dados e define como, a partir de uma relação (tabela, entidade), se pode buscar informações associadas e armazenadas em outra relação (tabela, relacionamento).

No DER da figura abaixo, os relacionamentos são dados pelos losangos e linhas conectados às entidades, que são representadas pelos retângulos.

Atributos (campos de dados) especiais que identificam as entidades serão usados para realizar a conexão entre elas. Isso é feito por meio das **chaves estrangeiras**, um conceito que estudaremos logo mais, em conjunto com as **chaves primárias**, das quais falamos há pouco.



[Vídeo adicional](#)

Atributos Chaves

Chaves são campos (atributos) que permitem organizar os registros, agrupá-los em conjuntos e identificá-los de forma exclusiva.

Dado que temos o conceito de atributo composto, uma chave pode ser definida a partir de uma composição de campos.

Uma chave única identifica um único registro (uma única linha), e uma chave não-única agrupa no mesmo conjunto uma relação de linhas que possuam valores iguais nesse campo chave.

Por exemplo:

RA de estudante na entidade Aluno – chave única – cada estudante possui seu RA exclusivo, sem repetições em linhas de outros alunos

Curso do estudante na entidade Aluno – chave não-única – pois vários alunos podem ter o mesmo valor de curso, já que um curso pode ter diversos alunos matriculados.

As chaves podem ser classificadas de acordo com sua unicidade ou não:

- Únicas: chaves candidatas, chaves compostas, chaves primárias, chaves incrementais
- Não-únicas: chaves estrangeiras (simples ou compostas)

Chave Candidata

Durante a modelagem, um atributo simples ou composto, que possua potencial de identificar univocamente uma instância de entidade, é chamada de chave candidata. Em outras palavras, ela poderá ser usada como chave primária se o processo de modelagem assim o requerer.

Uma chave candidata que não seja usada como a chave primária oficial será conhecida como **Chave Alternativa**. Em algumas situações, chaves alternativas podem ser a base para a criação de um objeto de banco de dados chamado Índice, objeto esse que permite buscar dados com grande rapidez, sendo essa busca feita através da chave alternativa ou da chave primária.

Como exemplo, temos o RA e o CPF de um aluno na entidade Aluno, ou o número de matrícula, RG ou CPF de um funcionário na entidade Funcionario.

Elas poderão ser úteis no processo de Normalização da Forma Normal de Boyce-Codd.

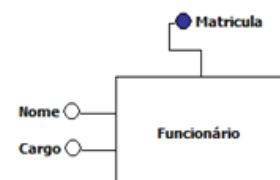
Chave Primária

Dentre as chaves candidatas que a modelagem for detectando, será escolhida uma como chave principal nessa relação.

Ela identifica de maneira unívoca (exclusiva, sem repetições) cada registro de uma relação. Também não pode ter valores nulos, pois é um atributo de preenchimento obrigatório.

Como a maioria dos sistemas de bancos de dados utiliza termos em inglês, podemos chamar as chaves primárias de Primary Key ou, abreviando, **PK**.

Nos diagramas, as chaves primárias podem ser identificadas como um círculo preenchido, com o nome do atributo ao lado, ligado à entidade por uma linha.



Chave Estrangeira (Foreign Key – FK)

Uma chave estrangeira é um campo de uma tabela que remete à chave primária de outra tabela. Usamos chaves estrangeiras quando queremos que o valor de um campo seja validado a partir de um campo semelhante em outra tabela. Criamos assim uma relação de dependência (um relacionamento) entre duas tabelas.

Por exemplo, antes de informarmos em qual departamento um funcionário trabalha (na tabela de Funcionários), precisamos garantir que esse departamento existe na tabela de Departamentos. Assim, como vemos na figura abaixo, o CodDept da tabela Funcionario é uma chave estrangeira, que remete (se refere) ao campo CodDept da tabela Departamento:

chave estrangeira (FK - Foreign Key)			
Departamento		Funcionario	
CodDept	Nome	CodFunc	Nome
D001	Financeiro	1001	Antonio
D002	Engenharia	1002	Beatriz
D003	Comercial	1003	Cláudio
			CodDept
			D002
			D003
			D001

Já o campo CodDept da tabela Departamento é a **chave primária** (primary key – PK) dessa tabela.

Observe que, no Modelo Relacional de Bancos de Dados, é a partir das chaves estrangeiras que conseguimos associar fisicamente uma tabela com outra. As operações relacionais buscam campos de chaves estrangeiras e os comparam com as chaves primárias da outra tabela para juntar os dados de uma tabela com dados de outra.

O relacionamento é estabelecido pela conexão entre a chave estrangeira de uma tabela com a chave primária da outra tabela.

De certa maneira, podemos ver que a tabela Funcionario depende funcionalmente da tabela Departamento. A tabela Funcionario é considerada uma **entidade fraca** e a de Departamento, uma **entidade forte**, pois um funcionário, para existir no banco de dados, precisa estar associado a um departamento. Em outras palavras, a existência de um funcionário depende da existência prévia de um departamento e, nesse sentido, a tabela Departamento é uma entidade forte em relação à entidade fraca, Funcionario.

Nesse caso, a chave estrangeira CodDept da tabela Funcionario remete à chave primária da tabela Departamento. Dessa maneira as duas tabelas se conectam, se relacionam.

Em sistemas de bancos de dados, as chaves estrangeiras são conhecidas como Foreign Keys ou FKs.

Um valor de chave estrangeira deve corresponder a um valor existente da chave primária associada na entidade forte, ou valer null.

Chave Composta

Uma chave é composta quando é resultado da junção entre dois ou mais atributos. Em geral, elas são usadas quando não se pode identificar os registros de maneira exclusiva com uma única coluna.

Nesse caso, se usam diversos campos para, compostos, fazer o papel de chave primária e, assim, permitirem identificar de forma exclusiva cada registro de uma entidade. Veja a tabela RendimentoEscolar, abaixo, como exemplo. Ela relaciona quais alunos, de quais cursos, estão matriculados em cada disciplina, e define nota, frequência e situação para cada trio dessas chaves estrangeiras:

codCurso	codDisciplina	RA	nota	frequencia	situação
28	TI101	20202	3,5	70	retido
28	TI101	20205	7,0	100	aprovado
28	TI102	20202	8,0	79	aprovado
28	TI102	20205	6,5	05	aprovado
39	DS101	20456	8,8	85	aprovado
39	DS101	20673	7,0	65	retido
39	DS102	20456	9,5	100	aprovado
59	DS102	20728	6,5	85	aprovado
59	DS102	20805	7,5	90	aprovado
59	DS105	20673	5,8	98	aprovado

Nela, apenas o campo CodCurso não poderia ser chave primária, pois há várias repetições de código de curso. Da mesma maneira, codDisciplina também possui repetições, assim como RA. Para que haja uma chave primária, podemos então agrupar (compor) os campos codCurso, codDisciplina e RA para que, tratados em conjunto, identifiquem univocamente cada linha.

Observe que, quando fazemos isso, teremos os dados ao lado tratados como chave, e que não haverá repetições:

Você define no modelo e no diagrama que esses campos formam a chave primária e, no momento de gerar a tabela no sistema de banco de dados, haverá uma maneira para indicar a composição de chaves, algo como “Primary Key (codCurso, codDisciplina, RA)”.

28TI10120202
28TI10120205
28TI10220202
28TI10220205
39DS10120456
39DS10120673
39DS10220456
59DS10220728
59DS10220805
59DS10520673

Chave Incremental, Surrogada ou Substituta

Uma chave surrogada (no inglês, surrogate key), é um atributo que o modelador do banco de dados adiciona a uma tabela e substitui campos chaves candidatas por um valor numérico, único para cada registro, e que assume o papel de chave primária no lugar de um campo como RA, CPF ou código Produto.

Esse atributo raramente será citado pelos usuários no momento da análise de requisitos, pois os usuários, normalmente, não têm ideia sobre campos chave e, ainda menos, sobre a possibilidade de chave substituta numérica.

Elas também podem ser chamadas de chaves incrementais, pois, geralmente, a cada novo registro incluído no banco de dados, mecanismos internos incrementam sequencialmente o valor dessa chave em relação ao valor do último registro incluído anteriormente. Em outras palavras, automaticamente esses campos são incrementados a cada novo registro incluído.

Os valores dessas chaves, normalmente, não são conhecidos dos usuários, nem mesmo são exibidos pelas aplicações quando os usuários finais acessam as aplicações. São controladas internamente.

Abaixo temos um exemplo com as tabelas Curso, Aluno, Disciplina e uma associação entre elas, que é a tabela Rendimento. Note que as 3 últimas tabelas têm uma **chave primária incremental**, idAluno, idDisciplina e idRendimento. Na tabela Rendimento, idDisc e idAluno são *chaves estrangeiras*, que remetem (fazem associação, relacionam) aos registros específicos de alunos e de disciplinas que são referenciados por esses números. Nem todas as ligações foram mostradas por linhas e setas para não poluir demais a figura.

Curso

<u>codCurso</u>	nomeCurso	turnoCurso	tipoCurso
28	Informática	Matutino	Integrado
39	Desenvolvimento de Sistemas	Vespertino	Concomitante Externo
59	Desenvolvimento de Sistemas	Noturno	Concomitante Externo

Rendimento

<u>idRendimento</u>	<u>codCurso</u>	<u>idDisc</u>	<u>idAluno</u>	nota	frequencia	situação
1	28	1	1	3,5	70	retido
2	28	1	2	7,0	100	aprovado
3	28	2	1	8,0	79	aprovado
4	28	2	2	6,5	05	aprovado
5	39	4	3	8,8	85	aprovado
6	39	4	4	7,0	65	retido
7	39	3	3	9,5	100	aprovado
8	59	5	5	6,5	85	aprovado
9	59	5	6	7,5	90	aprovado
10	59	8	4	5,8	98	aprovado

Aluno

<u>idAluno</u>	R.A	nome	outros
1	20202	Ana Castro	...
2	20205	Célio Silva	...
3	20456	Aderbal Aleandro	...
4	20673	Eliseu Cintra	...
5	20728	Selena Gomes	...
6	20805	Nicole Jardins	...

Disciplina

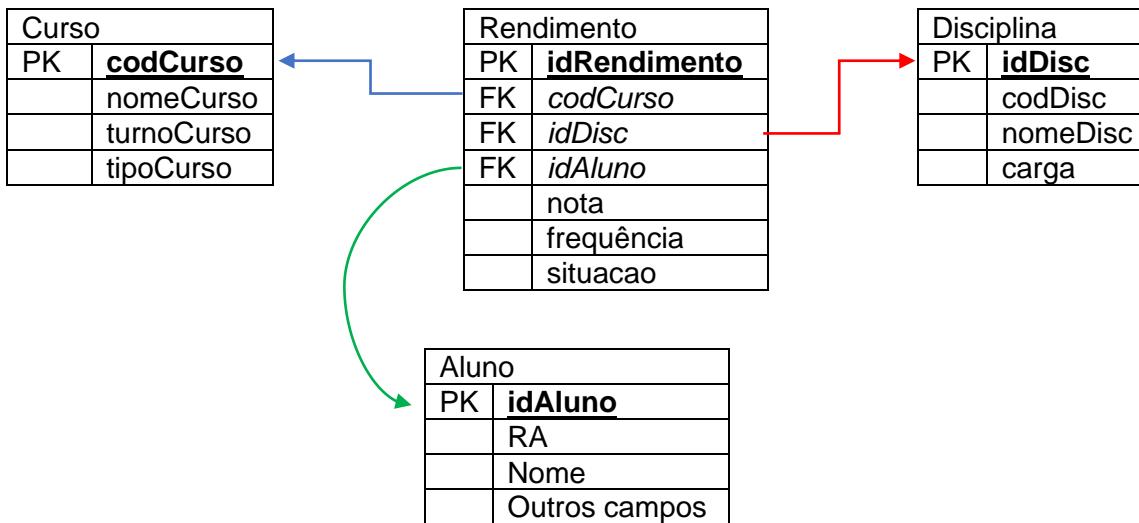
<u>idDisc</u>	CodDisc	nomeDisc	carga
1	TI101	TecPro 1	105
2	TI102	BanDad 1	45
3	TI103	DesInt 1	45
4	DS101	TecPro 1	105
5	DS102	BanDad 1	45
8	DS105	IntJogos	30

Observe que, nesse exemplo, por alguma decisão de modelagem, não se usou chave estrangeira incremental no campo codCurso da tabela Rendimento, usou-se diretamente como chave estrangeira o valor da chave primária da tabela Curso:

Na figura acima podemos verificar que as chaves estrangeiras funcionam como apontadores entre os registros; isso ocorre independentemente de a chave primária que apontam serem chaves incrementais ou não.

Representação das tabelas e suas associações pelas chaves primárias e estrangeiras.

As tabelas da figura acima possuem informações fictícias para que seja possível entender como os dados são armazenados e como as diferentes tabelas são associadas entre si. No entanto, durante a modelagem, podemos representar exatamente essa situação sem os dados, usando os atributos e tabelas e demonstrando suas associações, como vemos na próxima figura:



A definição de como essas tabelas ficarão, se terão chaves incrementais ou não, como serão divididas em duas ou mais tabelas são decisões que serão tomadas durante o processo de modelagem, em fases mais avançadas, com a normalização.

[Vídeo adicional](#)

Convenções para descrição de Entidades, Atributos e Relacionamentos

- **Entidades:** devem ter nome único, no singular, iniciando com letra maiúscula e sem espaços ou caracteres especiais. Procurar não usar nomes compostos, por exemplo, evitar nomes como DadosAluno. Não repetir nomes de entidades no mesmo banco de dados;
- **Atributos:** nome no singular, iniciando com letras minúsculas; no diagrama, atributos multivvalorados devem ser marcados com um “*” e o campo **chave primária** deve ser marcado com um “#” ou círculo preenchido. Não repetir nomes de atributos dentro da mesma entidade;
- **Relacionamentos:** usaremos como identificador do relacionamento um verbo, preferencialmente, pois relacionamento indica atividade (por exemplo, trabalha-em, compra, coordena, é-matriculado-em); indicamos sua opcionalidade (“deve ser” ou “pode ser”) e seu grau (binário, ternário, etc.), sua **cardinalidade** (“um e apenas um” ou “um ou mais”, “um para um”, “um para muitos”).

Cardinalidade

Número (mínimo, máximo) de ocorrências de entidade associadas a uma ocorrência da entidade em questão através do relacionamento.

Cardinalidade mínima

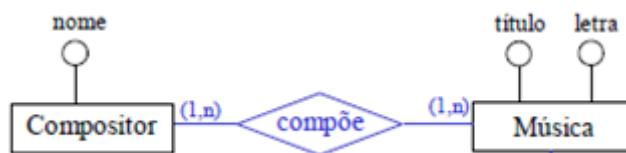
É o número mínimo de ocorrências de entidade que são associadas a uma ocorrência da mesma (auto-relacionamento) ou de outra(s) entidade(s) através de um relacionamento.

A cardinalidade mínima 1 recebe a denominação de **associação obrigatória**, já que ela indica que o relacionamento deve obrigatoriamente associar uma ocorrência de entidade a outra. A cardinalidade mínima 0 (zero) recebe a denominação de **associação opcional**.

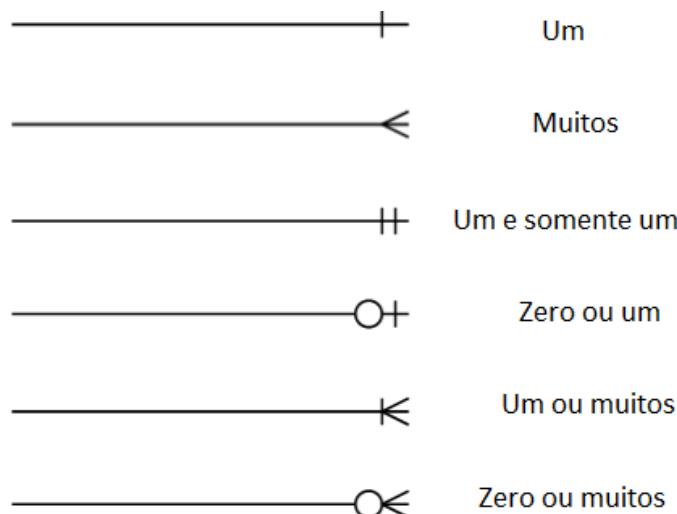
Cardinalidade máxima

É o número máximo de ocorrências de entidade que são associadas a uma ocorrência da mesma ou de outra entidade através de um relacionamento. Apenas duas cardinalidades máximas são relevantes: a cardinalidade máxima 1 e a cardinalidade máxima n (muitos).

No diagrama ER, indicamos cardinalidades mínima e máxima separadas por vírgula, **opcionalmente** entre parênteses, ao lado de cada entidade do relacionamento. Por exemplo:



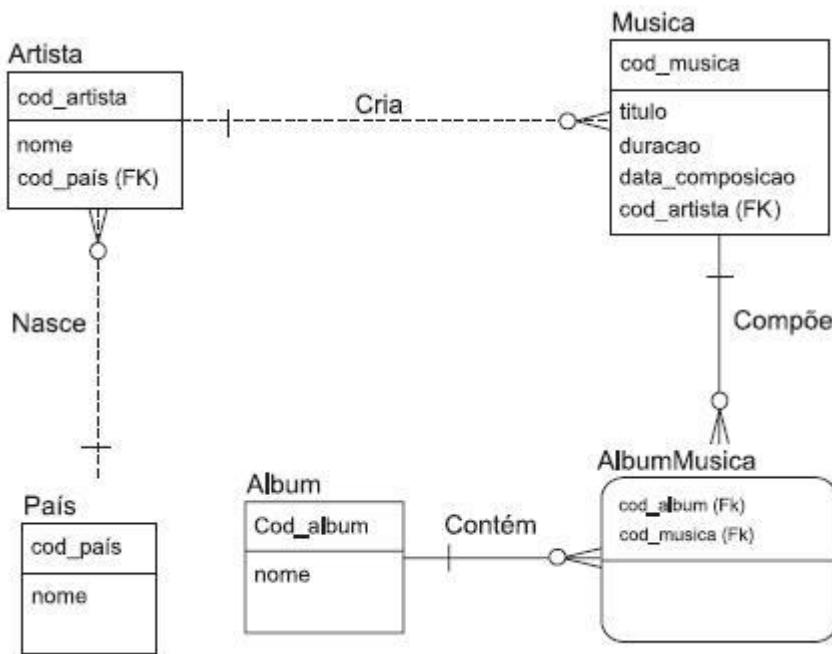
Essa é a notação originalmente sugerida por Peter Chen. Existe uma outra notação, chamada popularmente de “pé de galinha”, que tem as indicações abaixo:



No diagrama abaixo, da esquerda para a direita, lemos: um e somente um cliente solicita uma ou muitas encomendas. Da direita para a esquerda, uma ou muitas encomendas foram solicitadas por um único cliente.



Um outro exemplo:



[Vídeo adicional](#)

Tipos de Relacionamento

De acordo com a cardinalidade existem alguns tipos básicos de relacionamentos entre as entidades.

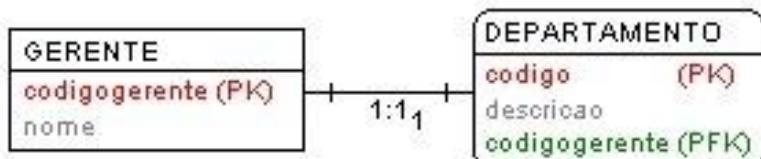
- RELACIONAMENTOS UM PARA UM
- RELACIONAMENTOS UM PARA MUITOS
- RELACIONAMENTOS MUITOS PARA MUITOS
- AUTO-RELACIONAMENTOS

Relacionamento Um para Um (1:1)

São relacionamentos em que uma ocorrência de uma entidade em A está associada no máximo a uma ocorrência em uma entidade B e uma ocorrência na entidade B está associada no máximo a uma ocorrência na entidade A.

Neste relacionamento, escolhemos qual tabela irá receber a chave estrangeira, e para cada valor do campo na tabela A, há no máximo um valor na tabela B.

No exemplo mostrado na figura abaixo podemos entender melhor este tipo de relacionamento, onde estaremos definindo que um Gerente (e somente um) gerencia um (e somente um) Departamento. Ou seja, o mesmo Gerente não pode gerenciar mais de um Departamento e um Departamento não poderá ser gerenciado por mais de um Gerente.



Relacionamento Um para Muitos (1:N)

Um relacionamento 1:n ocorre com frequência em situações de negócio. Às vezes ocorre em forma de árvore ou em forma hierárquica. No exemplo abaixo, temos a seguinte representação: cada curso cadastrado possui vários alunos ligados a ele, pois cada aluno, ao ser cadastrado, deverá ser ligado a um curso obrigatoriamente. O campo **codigoCurso** foi escolhido como **chave primária** na entidade CURSO, ou seja, ele não poderá se repetir dentre os vários registros de cursos. Já na tabela ALUNO, a chave primária é matrícula e o **codigoCurso** é o que chamamos de **chave estrangeira**. A representação ficaria assim:



Como lemos este relacionamento:

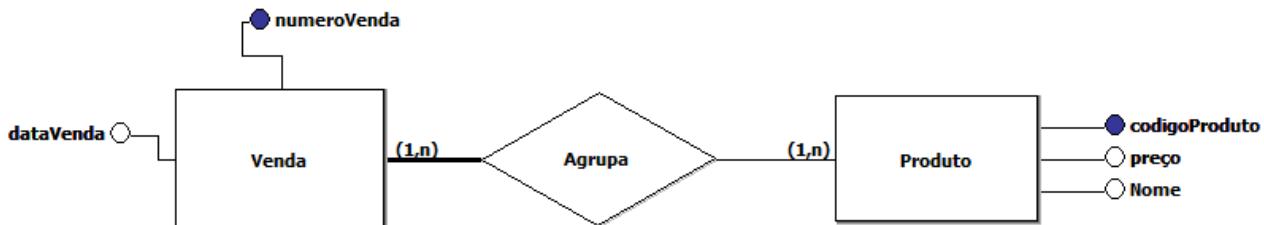
- **UM Curso matricula MUITOS Alunos**
- **UM Aluno se matricula em UM Curso**

Relacionamento Muitos para Muitos (N:N)

Uma ocorrência de uma entidade em A está associada a qualquer número de ocorrências na entidade B, e cada ocorrência da entidade em B está associada a qualquer número de ocorrências na entidade A.

Considere o caso em que produtos são vendidos. Podemos identificar imediatamente duas entidades: VENDA e PRODUTO. Uma venda pode ser composta por muitos produtos (mercadorias) e um produto pode aparecer em muitas vendas. Não estamos dizendo que um mesmo item de produto possa ser vendido muitas vezes, mas que o tipo específico de produto (por exemplo, um livro) pode ser vendido muitas vezes; temos, portanto, um relacionamento de muitos-para-muitos (n:n) entre VENDA e PRODUTO.

Observe a figura abaixo. Observe a representação do relacionamento. Cada uma das linhas que aparece no formulário do pedido de vendas é, em geral, conhecida no varejo como um item de linha, onde o código do produto é ligado a uma venda.



A representação desse relacionamento n:n é mostrada na figura acima. Dizemos muitos para muitos porque há dois relacionamentos: um mesmo código Produto poderá estar relacionado com muitas VENDAS e uma VENDA poderá estar relacionada com muitos códigoProdutos.

Como lemos esse relacionamento:

- **UMA VENDA POSSUI VÁRIOS PRODUTOS**
- **CADA PRODUTO PODERÁ ESTAR LIGADO A VÁRIAS VENDAS**

Por que criaremos uma terceira entidade?

Quando temos um relacionamento n:n e precisamos manter informações sobre este relacionamento, criamos uma entidade associativa para armazenar informações sobre o relacionamento. Neste caso, armazenamos dados sobre as mercadorias vendidas. Não podemos armazenar estes dados em VENDA, pois uma venda pode ter muitos itens e uma entidade só armazena ocorrências de valores simples. Da mesma maneira, não podemos armazenar esses dados em PRODUTO, porque um código de mercadoria pode aparecer em muitas vendas.

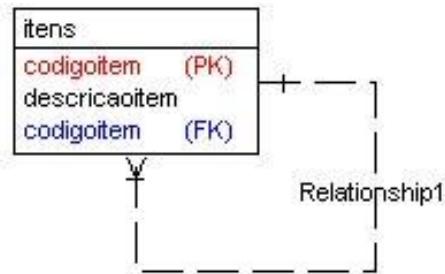
No modelo relacional, quando implantado fisicamente, será necessário criar uma terceira tabela que faça a ligação entre as duas tabelas principais. Ou seja, não é possível implantar esse conceito diretamente, pois o modelo relacional usa tabelas para armazenar os dados e, por isso, relacionamentos complexos (N:N) precisam de uma tabela para que seus dados sejam armazenados. Essa terceira tabela é chamada de tabela de associação ou tabela de detalhes. Ela deverá possuir uma chave primária composta de dois campos e chaves estrangeiras provenientes das duas tabelas originais. Assim, um relacionamento muito-para-muitos do modelo lógico deverá ser convertido para dois relacionamentos um-para-muitos com uma terceira tabela.

Relacionamentos Recursivos ou Auto-relacionamentos

Os relacionamentos recursivos (também chamados de auto-relacionamentos) são casos especiais onde uma entidade se relaciona com si própria. Apesar de serem relacionamentos muito raros, a sua utilização é muito importante em alguns casos.

Os auto-relacionamentos podem ser do tipo 1:1 (um-para-um), 1:N (um-para-muitos) ou N:M (muitos-para-muitos), dependendo da política de negócio que estiver envolvida.

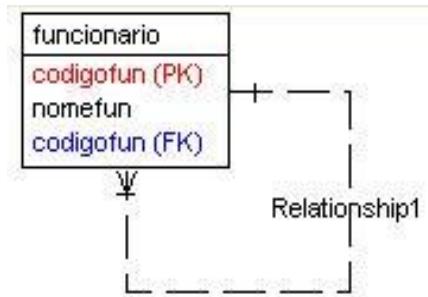
Exemplos deste relacionamento podem ser encontrados na chamada “explosão de materiais”, onde itens compostos são formados por muitos itens componentes; por sua vez, estes itens compostos podem ser componentes de outros itens maiores. Exemplificando, temos um automóvel, que é composto pelo chassis, motor, direção, câmbio etc.; O motor, por sua vez, é formado pelo carburador, velas, platinado etc. Esta explosão pode ser representada pelo seguinte relacionamento:



ITEM (N) compõe (M) ITEM

sendo que o papel do ITEM é ora de componente e ora de composto.

Um outro exemplo de auto-relacionamento é o gerenciamento de funcionários, onde o gerente é um funcionário que possui um relacionamento com outros funcionários que lhe são subordinados. Este relacionamento pode ser representado da seguinte forma:



FUNCIONÁRIO (1) gerencia (N) FUNCIONÁRIO

sendo que o papel do FUNCIONÁRIO é ora de gerente e ora de subordinado.

[Vídeo adicional](#) (6:26 minutos para explicação adicional sobre tipos de relacionamentos)

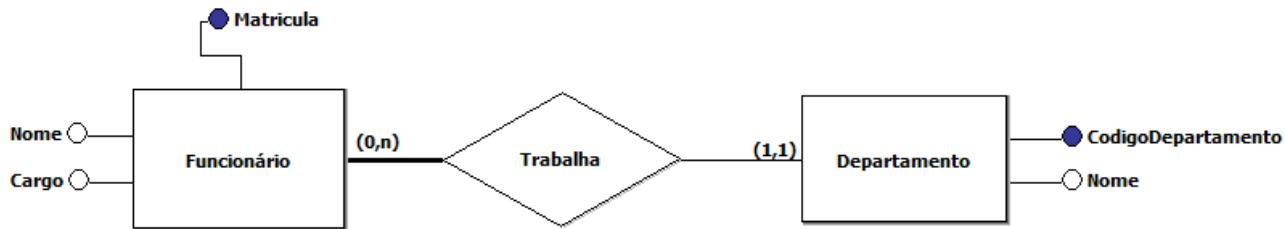
Outros componentes de um Banco de Dados

Há vários outros componentes em bancos de dados, que estudaremos neste e no próximo semestre, como índices, procedimentos e funções armazenados, gatilhos, regras, dentre outros.

Criando um Modelo Lógico

O modelo lógico possui conceitos que os usuários são capazes de entender, ao mesmo tempo em que não está distante do modelo físico do banco de dados, embora ainda seja independente de qualquer SGBD.

Já conhecemos os tipos de dados e definimos as Entidades, Atributos e Relacionamentos no Diagrama Entidade Relacionamento. Por exemplo:



Acima, temos um diagrama criado com a ferramenta **brModelo**. Ela define duas entidades (Funcionário e Departamento) e um relacionamento (Trabalha). Há também os campos de identificador único (matricula e codigoDepartamento), além de outros atributos de cada entidade (nome, cargo de Funcionário e nome de Departamento). O relacionamento indica a cardinalidade, que são os números entre parênteses. Eles indicam quantidades relativas entre as entidades. Por exemplo, de 0 a N (muitos) funcionários trabalham em um único departamento (1,1). Pode-se também ler esse relacionamento ao contrário. Em um Departamento trabalham de 0 a N funcionários.

EXERCÍCIOS

Elabore o diagrama de relacionamentos com cardinalidade e atributos para cada caso abaixo:

1. Refaça os exercícios da [seção 3.3.1](#), elaborando o modelo ER de cada caso.
2. Uma universidade tem muitos estudantes e um estudante pode se dedicar a no máximo uma universidade.
3. Uma aeronave pode ter muitos passageiros, mas um passageiro só pode estar em um voo de cada vez.
4. Uma nação possui vários estados, e um estado, muitas cidades. Um estado só poderá estar vinculado a uma nação e uma cidade só poderá estar vinculado a um estado.
5. Um encontro de eventos esportivos pode ter muitos competidores e um competidor pode participar de mais de um evento.
6. Um paciente pode ter muitos médicos e um médico muitos pacientes.
7. Um aluno pode frequentar mais de uma disciplina e uma mesma disciplina pode ter muitos alunos.
8. Um aluno pode realizar um ou mais projetos. Um projeto é realizado por um ou mais alunos.
9. Um diretor dirige no máximo um departamento. Um departamento necessariamente tem no máximo um diretor.
10. Um autor escreve um ou mais livros. Um livro pode ser escrito por vários autores.
11. Uma equipe de futebol de campo é composta por vários jogadores. Um jogador joga apenas em uma equipe.
12. Um cliente realiza nenhuma ou várias encomendas. Uma encomenda diz respeito apenas a um cliente.

Exercícios adicionais da apostila 2011 prof. Marcos Alexandruk:

13. Várias empresas possuem frotas de veículos que são identificados através da placa (XYZ-1234). São registrados também os fabricantes e modelos de cada veículo. Os funcionários são identificados através do número de matrícula. São mantidos registros do nome e CPF de cada funcionário. Criar o MER (Modelo Entidade-Relacionamento) para cada um dos casos descritos a seguir:

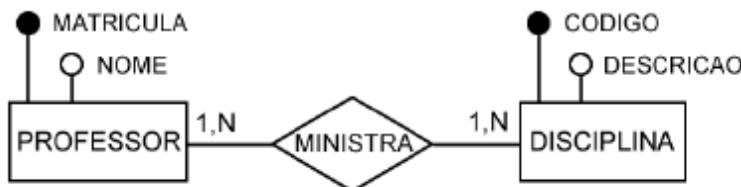
- Empresa A: Cada veículo (sem exceção) é dirigido por um apenas funcionário. Todos os veículos estão alocados aos funcionários. Cada funcionário pode utilizar apenas um veículo e todos os funcionários têm veículos pertencentes à frota da empresa.
- Empresa B: Cada veículo (sem exceção) é dirigido por um apenas funcionário. Todos os veículos estão alocados aos funcionários. Cada funcionário pode utilizar apenas um veículo, porém alguns funcionários não têm veículos pertencentes à frota da empresa.
- Empresa C: Cada veículo pode ser dirigido por um ou mais funcionários. Todos os veículos estão alocados aos funcionários. Alguns funcionários podem utilizar mais um veículo e todos os funcionários têm veículos pertencentes à frota da empresa.
- Empresa D: Cada veículo (sem exceção) é dirigido por um apenas funcionário. Todos os veículos estão alocados aos funcionários. Alguns funcionários podem utilizar mais de um veículo, porém alguns funcionários não têm veículos pertencentes à frota da empresa.
- Empresa E: Cada veículo pode ser dirigido por um ou mais funcionários. Todos os veículos estão alocados aos funcionários. Cada funcionário pode utilizar apenas um veículo e todos os funcionários têm veículos pertencentes à frota da empresa.
- Empresa F: Alguns veículos podem ser dirigidos por mais de um funcionário. Porém, outros veículos não podem ser funcionários. Cada funcionário pode utilizar apenas um veículo e alguns funcionários não têm veículos pertencentes à frota da empresa.

14. Tomando como base os modelos a seguir elabore um texto breve (similar aos apresentados nas questões acima) para explicar cada caso.

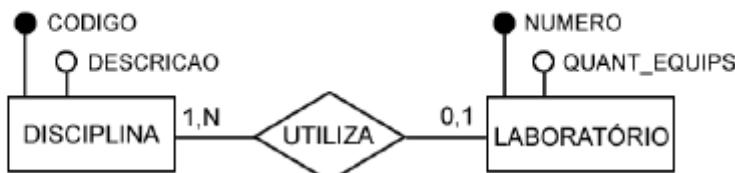
a.



b.



c.



TRABALHO DE MODELAGEM DE DADOS

Objetivo

Sabendo-se da importância de se modelar um banco de dados é imprescindível antes de sua implantação, temos traçado como objetivo deste trabalho, a criação de um projeto de banco de dados banco de dados que servirá a um determinado seguimento comercial, industrial ou educacional.

Etapas do Trabalho

1 – Tema

O tema deverá abordar modelos de negócios reais e será entregue pelo professor para os alunos. Cada grupo terá um tema diferente.

2 – Análise do projeto de banco de dados (detalhes na aula 3)

- Determinar quais os requisitos de informação que serão atendidos pelo banco de dados
- Determinar as entidades
- Determinar os atributos de cada entidade
- Determinar os relacionamentos entre as entidades do modelo
- Demonstrar o dicionário de dados descrevendo cada entidade do modelo com relação a atributos, tipos de dado, obrigatoriedade, chave primária, regras de negócio e restrições de integridade.

3 – O trabalho deverá ser feito em formato PDF, retratando passo a passo da elaboração do projeto do banco de dados, seguindo as normas da ABNT estabelecidas de acordo com a orientação para trabalhos acadêmicos. Deverá haver uma capa com matrícula e nome dos componentes do grupo, e uma página de apresentação do trabalho, com uma breve introdução falando sobre a importância de modelar um banco de dados antes de sua implantação. Na última página deverá estar uma breve conclusão.

4 – O trabalho deverá ser feito em grupo de até no máximo 3 pessoas.

5 – Pontuação (0 – 4):

6 – **A data limite para envio de todos os trabalhos será a data da primeira avaliação**

3.3.3. Restrições de Integridade

Um sistema de banco de dados precisa garantir a consistência e precisão dos dados, para que sejam confiáveis, corretos e disponíveis.

A definição das regras que garantem esses fatores é feita durante o processo de modelagem e implementação dos sistemas de banco de dados, através da criação e aplicação de Restrições de Integridade.

Há vários tipos de Restrições de Integridade, dentre as quais podemos destacar:

- Integridade de Domínio
- Integridade de Vazio
- Integridade de Chave
- Integridade Referencial
- Integridade definida pelo usuário

Vamos discutir um pouco de cada uma:

Integridade de Domínio

Está relacionada à declaração dos atributos de uma entidade. Quando pensamos nos atributos, devemos pensar também na natureza de cada um deles, ou seja, que tipos de dados cada atributo possui, bem como qual tamanho máximo ocupa no banco de dados.

Essa natureza dos dados, definida pelo tipo, tamanho, valores mínimo e máximo, dentre outros fatores, é chamada de **domínio**.

Há vários tipos de dados, como inteiro, real, cadeia de caracteres de tamanho fixo, cadeia de caracteres de tamanho variável, datas, valores lógicos, textos, imagens, e tipos binários, que permitem armazenar qualquer tipo de dado, por exemplo um vídeo ou documento PDF.

Cada tipo de dado possui uma representação interna, que pode variar de sistema para sistema de banco de dados; por exemplo, quantidade de bytes que um inteiro ocupa pode variar de banco para banco.

O tipo de um atributo determina quais valores podem ser aceitos naquele atributo e também, por exclusão, quais valores não podem ser aceitos. Por exemplo:

NotaAluno: valor real

Valor permitido:	10.0	9.5	0.0	3.72	12.7	-7,23	(pois é valor real)
Valores não permitidos:	'Maria'		'10.0'		'Cinco e meio'		false

SiglaUF : char(2)

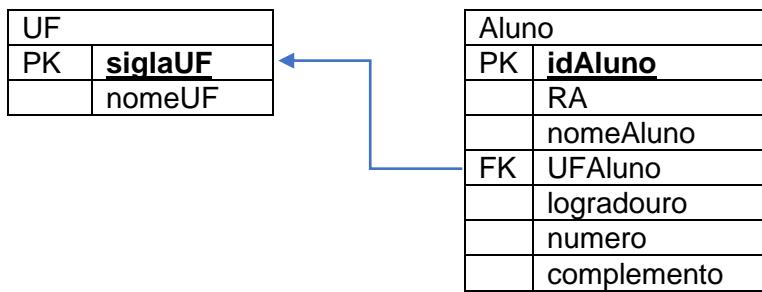
Valor permitido:	'XY'	'SP'	'BA'	'CE'	'RJ'	'IX'	'7P'
Valores não permitidos:	'Maria'		10	-5		true	

Integridade Referencial

Este tipo de integridade assegura que um atributo de uma tabela A somente aceite valores que existam em outra tabela B. Em outras palavras, para que se possa incluir um registro na tabela A, os valores de certos atributos deverão existir previamente em outro atributo da tabela B.

Portanto, A e B são tabelas relacionadas entre si e A possui um atributo que é chave estrangeira e **referencia** a chave primária da tabela B. Esse atributo da tabela A não poderá ser preenchido com valores que já não estejam gravados previamente entre os valores de chave primária da tabela B.

Por exemplo, se na tabela Aluno temos o atributo UFAluno, que seria a sigla de uma Unidade da Federação do Brasil, e temos que existe um relacionamento entre a tabela Aluno e a tabela UF, cuja chave primária é SiglaUF, somente serão aceitos em UFAluno valores que existam no campo SiglaUF da tabela UF:



Aluno				
<u>idAluno</u>	RA	NomeAluno	UFAluno	...
10	19378	Célia Shirlei	SC	
12	20212	Adailton Sé	CE	
13	20310	Marta Souza	SP	
14	20401	Luis Felipe	SP	
17	20408	Railton Cruzes	AM	
18	20425	Cláudia Simões	SP	
19	20809	Igor Selles	MG	

UF	
<u>siglaUF</u>	nomeUF
AM	Amazonas
BA	Bahia
CE	Ceará
SC	Santa Catarina
SP	São Paulo
RJ	Rio de Janeiro

Na tabela de Alunos acima, não podemos incluir o registro de idAluno 19, porque a UFAluno igual a MG não existe na tabela UF (campo siglaUF). Dessa forma, ao se fazer a referência a MG na tabela UF, o sistema de banco de dados acusa erro, impedindo a inclusão desse registro. Isso é feito pelo sistema automaticamente, depois que as duas tabelas tiverem sido criadas e esse relacionamento entre elas for definido.

Tentar incluir um registro com um dos atributos violando a integridade referencial impede a inclusão.

Tanto a garantia da integridade quanto o impedimento de inclusão no caso de violação da integridade já vêm previamente programados no sistema de banco de dados e, dessa forma, o controle dessa situação não precisa ser programado nas aplicações que acessam os dados no banco de dados. Apenas é necessário tratar as situações em que o sistema de banco de dados avisa a aplicação de que não foi possível incluir. Nesse caso, a aplicação deveria emitir uma mensagem ao usuário que solicitou a inclusão informando-o de que houve uma violação de integridade referencial. Pode-se mesmo informar que MG não é uma Unidade da Federação previamente existente.

Da mesma maneira, se você tentar excluir um registro da tabela de Unidades da Federação, e esse registro estiver relacionado com um registro da tabela de Alunos, a exclusão não poderá ser feita ou, conforme for feita a configuração do banco de dados, todos os alunos daquela UF serão eliminados também. Isso é chamado de exclusão em cascata.

Por exemplo, imagine que queremos excluir o registro de São Paulo, cuja PK é 'SP'. Se você configurar a tabela de UF para que haja exclusão em cascata, ao excluir o registro de SP na tabela de UF, os registros dos RAs 20310, 20401 e 20425 serão excluídos automaticamente pois, para manter a integridade referencial, o sistema de banco de dados apaga os registros de alunos que violariam a integridade.

Por outro lado, se você configurar a tabela de UF para que não haja exclusão em cascata, ao tentar excluir o registro de siglaUF = 'SP', o sistema de banco de dados impede a exclusão desse registro nessa tabela, pois isso faria com que o banco não ficasse íntegro, já que haveria pelo menos 3 registros na tabela Aluno que ficariam "órfãos" de UF.

Integridade de Vazio

Este tipo de integridade define se um atributo é obrigatório ou opcional. Se for obrigatório, sempre se terá de colocar um valor válido nessa coluna. Se for opcional, essa coluna poderá ficar vazia, ou seja, com valor “nulo” (**null**).

Null é diferente de zero ou de espaços em branco. Null informa ao sistema de banco de dados que não foi colocada nenhuma informação naquele campo e isso pode ser usado nas aplicações para solicitar, aos usuários, complementação dos dados ou, simplesmente, não fazer nada. Depende do que se deseja fazer com esse campo e com as regras de negócio que usam o banco de dados.

Alguns campos, por natureza, não podem ficar nulos. Por exemplo, chaves primárias, campos únicos, ou o nome de um aluno, por exemplo. Já o telefone do aluno poderá ser opcional (aceitar nulo), pois nem todo aluno tem telefone.

A definição de quais campos podem ser ou não vazios depende da Análise de Requisitos, e fará parte da modelagem do banco de dados.

Integridade de Chave

Os valores incluídos na coluna de chave primária nunca poderão ser repetidos entre os vários registros de uma tabela, e também não poderão ser nulos.

Dessa maneira, as linhas nunca serão idênticas entre si pois, mesmo que todos os demais atributos sejam idênticos, a coluna de chave primária os diferenciará.

Se você tentar incluir um registro em uma tabela e, por acaso, o valor da chave primária que você digitar já exista nessa tabela (em alguma outra linha de dados, na coluna da chave primária), o sistema de banco de dado impedirá a inclusão. Essa funcionalidade também já vem programada no sistema e facilita a vida dos programadores de aplicações que acessam o banco, pois não terão de pesquisar por repetições de chaves primárias antes de incluir um registro.

Integridade definida pelo usuário

Os usuários do banco de dados, durante a modelagem, poderão definir regras de negócio específicas e que determinem que certas colunas só possam assumir valores dentro de algum intervalo significativo para essas regras de negócio ou, por exemplo, apenas valores que passem por algum cálculo matemático anterior.

Essas regras são implementadas durante a criação do banco de dados físico, mas é importante que sejam pensadas e definidas antes dessa fase.

[Vídeo adicional](#)

3.3.4. Diagrama Entidade-Relacionamento

Já estudamos como os diversos elementos do Modelo Entidade-Relacionamento são apresentados, de forma gráfica, pelo Diagrama ER.

No vídeo abaixo, há uma aula revisando esse assunto.

[Vídeo sobre Diagrama Entidade-Relacionamento](#)

3.3.5. Dicionário de Dados

O Dicionário de Dados é um documento que descreve a estrutura de cada tabela do banco de dados. Essa estrutura envolve a natureza de cada atributo, seu tamanho e restrições previstas em termos de dados aceitos e não-aceitos. Também descreve os relacionamentos entre as tabelas.

No início desta disciplina, falamos sobre Metadados. O Dicionário de Dados é um metadado, pois ele descreve dados sobre os dados (sobre os atributos, tabelas, relacionamentos) que formam um banco de dados.

O Dicionário de Dados serve também como documento resultante da modelagem, pois as descrições que contém são como uma diretriz para a criação do modelo físico do banco de dados.

Nessa ótica, essa ferramenta é um auxiliar bastante importante do processo de desenvolvimento do banco de dados. Seu uso diminui a quantidade de erros futuros no processo de criação física do banco de dados, pois ficam registradas todas as definições de campos, seus formatos, tabelas, relacionamentos e, não menos importante, o significado e uso de cada um desses objetos componentes do banco.

Você usa o Diagrama Entidade-Relacionamento como base para criação do Dicionário de Dados. Relaciona todas as tabelas existentes, para que são usadas, seus relacionamentos e o que significam, cardinalidade, etc. Em seguida, descreve cada tabela, seus atributos, chaves primárias e estrangeiras,

Temos abaixo alguns exemplos de dicionários de dados descrevendo tabelas:

Entidade: Cliente				
Atributo	Classe	Domínio	Tamanho	Descrição
Código_cliente	Determinante	Numérico		
Nome	Simples	Texto	50	
Telefone	Multivvalorado	Texto	50	Valores sem as máscaras de entrada
Cidade	Simples	Texto	50	
Data_nascimento	Simples	Data		Formato dd/mm/aaaa

TABELA: CIDADES					
	CAMPO	DESCRIÇÃO	TIPO	TAM	DEC
PK	CID_CEP	Código de Endereçamento Postal	INTEIRO	8	-
	CID_NOME	Nome da Cidade ou Localidade	CARACTER	100	-
	CID_UF	Nome da Unidade Federativa	CARACTER	100	-

TABELA: USUÁRIOS					
	CAMPO	DESCRIÇÃO	TIPO	TAM	DEC
PK	USU_CODIGO	Código do Usuário	INTEIRO	35	-
FK	CID_CEP	Código de Endereçamento Postal	INTEIRO	8	-
	USU_NOME	Nome do Usuário	CARACTER	100	-
	USU_ENDEREÇO	Endereço do Usuário	CARACTER	100	-
	USU CPF	CPF do Usuário	CARACTER	100	-
	USU RG	Identidade do Usuário	CARACTER	100	-
	USU DATANASC	Data de Nascimento do Usuário	DATA	-	-

TABELA: Tb_Aluno		Cadastro de alunos				
CAMPO LÓGICO	CAMPO FÍSICO	TIPO	PK	FK (Tabela/Campo)	RESTRIÇÕES	OBSERVAÇÕES
Código	Alu_codigo	SMALLINT	PK		NÃO NULO E MAIOR QUE ZERO	Campo auto-incremento
Nome	Alu_nome	VARCHAR(100)			NÃO NULO	Informar se usuário incluir somente uma palavra.
Data de Nascimento	ento	DATE			Data mínima < HOJE	
Código da turma	Tur_codigo	SMALLINT		Tb_Turma/Tur_codigo	ZERO	
Sexo	Alu_sexo	CHAR(1)			Somente "M" ou "F"	M = Masculino / F = Feminino
Nome do pai	Alu_pai	VARCHAR(100)				
Nome da mãe	Alu_mae	VARCHAR(100)				
Nota Media	Not_codigo	SMALLINT		Tb_Acompanhamento Aluno/aco_codigo		
Registro do Aluno	Alu_RA	SMALLINT			NÃO NULO E MAIOR QUE ZERO	Campo auto-incremento
Informações complementares	Alu_Complemento	BLOB				
Telefone	Alu_telefone	INTEGER				Telefone para contato
Celular	Alu_celular	INTEGER				Celular para contato
Endereço	Alu_endereco	VARCHAR(150)				Endereço do aluno

Entidade						
Atributo	Nome do campo	Tipo de dado	Tamanho	Restrição	Descrição	
Código de categoria	cod_categoria	Numérico inteiro	10	Chave primária	Usado para distinguir as categorias	
Descrição da categoria	desc_categoria	Alfanumérico	40		Descreve a categoria do game, citando exemplos e	
Nome da categoria	nome_categoria	Alfanumérico	40	Não nulo.	Tem a função de	

Atributos						
Atributo	Nome do campo	Tipo de dado	Tamanho	Restrição	Descrição	
Código de categoria	cod_categoria	Numérico inteiro	10	Chave primária	Usado para distinguir as categorias	
Descrição da categoria	desc_categoria	Alfanumérico	40		Descreve a categoria do game, citando exemplos e	
Nome da categoria	nome_categoria	Alfanumérico	40	Não nulo.	Tem a função de	

Observe que não há um formato padronizado, e há algumas ferramentas computacionais que permitem criar dicionários de dados. Algumas dessas ferramentas, inclusive, podem ser usadas para criar as tabelas do banco de dados físico após o término da modelagem.

Temos abaixo um exemplo para uma Locadora de Video. Primeiramente descrevemos as tabelas e seus relacionamentos:

Tabela	Relacionamento	Nome do Relacionamento	Descrição
Filme	Categoria	Pertence	Catálogo de filmes do acervo da locadora
	Locação	Empréstimo	
Categoria	Filme	Agrupa	Tipos de categorias de filmes
	Locação	Aluga	
Cliente	Dependente	Autoriza	Titular de empréstimos de filmes
	Dependente	Autoriza	
Dependente	Cliente	Autorizado por	Autorizado a tomar filmes
Locação	Filme	Contém	Relação de filmes emprestados
	Cliente	Empresta	

Posteriormente, descrevemos os atributos de cada tabela (abaixo temos algumas como exemplo):

Tabela: Filme

Atributo	Tipo	Tamanho	Restrições	Valor Default	Descrição
IdFilme	Inteiro	4 bytes	PK, not null	Autoinc	Identificação do filme, gerado automaticamente
Titulo	Varchar	50	Not null		Título do filme
idDiretor	Inteiro	4 bytes	FK		Identificação do diretor do filme
idCategoria	Inteiro	4 bytes	FK, not null		Identificador da categoria do filme
Sinopse	Texto	Livre	Null		Resumo do filme
Imagen	Image	Livre	Null		Cartaz do filme

Tabela: Categoria

Atributo	Tipo	Tamanho	Restrições	Valor Default	Descrição
IdCategoria	Inteiro	4 bytes	PK, not null	Autoinc	Identificação da categoria, gerado automaticamente
Descricao	Varchar	50	Not null		Descrição da categoria

Tabela: Locação

Atributo	Tipo	Tamanho	Restrições	Valor Default	Descrição
idLocacao	Inteiro	4 bytes	PK, not null	Autoinc	Identificação da locação, gerado automaticamente
IdFilme	Inteiro	4 bytes	FK, not null		Identificação do filme alugado
idCliente	Inteiro	4 bytes	FK, not null		Identificação do cliente que alugou o filme
idDependente	Inteiro	4 bytes	FK, null		Identificador do dependente que retirou o filme sob autorização de um cliente
DevolucaoPrevista	Data	8 bytes	Not Null		Data de devolução prevista
DevolucaoEfetiva	Data	8 bytes	Null		Data de devolução efetiva
ValorPrevisto	Real	8 bytes	Not null		Valor a ser pago
multaPorAtraso	Real	8 bytes		0	

É importante definir os tipos de atributos antes de se criar o banco de dados físico. Essas informações serão necessárias no momento da criação das tabelas e relacionamentos e não podemos deixar para pensar nessas informações nesse momento, pois poderemos “chutar” tipos, tamanhos e restrições errados. Planejar antecipadamente sempre é muito importante, e o Dicionário de Dados faz parte desse planejamento.

Temos abaixo um exemplo mais completo, da UTFPR. Você pode consultar o link desse material para um conteúdo específico dessa universidade.

Nomenclatura do dicionário de Dados

Para cada tabela do banco de dados serão apresentadas três comentários sobre a mesma, sendo que devem compreender:

a) Entidade

Nome	Nome da entidade
Sigla	Sigla ou nome abreviado
Descrição	Descrição do conteúdo da Entidade no contexto do projeto, deixando evidente o que poderá ser registrado na entidade e o que não poderá estar, caso isto seja necessário.

b) Relacionamentos

Com a Entidade	Nome abreviado da entidade com a qual existe um relacionamento
Cardinalidade	Tipo do relacionamento, indicativo da cardinalidade do relacionamento, no formato x:y onde X = cardinalidade na entidade em descrição, podendo ser: 0, 1 ou M (muitos) Y = cardinalidade na outra entidade, podendo ser 0, 1 ou M
Atributo	Nome do atributo que estabelece o relacionamento, na entidade que está sendo descrita, ou na entidade relacionada.
Nome	Nome do relacionamento e sua descrição (o que representa no contexto do negócio).

c) Elementos de Dados

Nome	Nome do atributo, conforme será utilizado pelos programas e linguagem SQL.
Característica	Simples, composto, identificador, único, multivvalorado, derivado,...
Tipo	Tipo do Dado: Varchar2= Conjunto de caracteres N = Number D = Date.
Chave	Indicador de campo chave CP → chave primária CE → chave estrangeira CS → Chave secundária
Descrição	Descrição estendida do atributo. Todos os detalhes referentes ao atributo devem ser relacionados nesta coluna. Caso possua máscara de edição esta deve ser indicada nesta coluna, tais como: a) regras de validação b) valor padrão c) se pode ser nulo ou é requerido

Exemplo de Dicionário de Dados

Entidade:

Nome da Entidade: Tb_Autor	Sigla: Aut
Descrição: Refere-se aos dados dos Autores das obras cadastradas.	

Relacionamentos:

Com a Entidade	Cardinalidade	Atributo(s)	Nome do relacionamento e significado
Tb_AutorLivro	1:M	AutLiv_CodAutor	Chave Primária, representa o código do autor.

Elementos de dados:

Nome do Atributo	Característica	Tipo	Chave	Descrição estendida e observações
Aut_Codigo	Identificador	Number	CP	Refere-se ao código do autor. Campo não nulo.
Aut_Nome	Simples	Varchar2(60)	-	Representa o nome do autor. Campo não nulo.

Fonte: [Exemplo completo de Dicionário de Dados da UTFPR](#)

[Ferramenta computacional para Dicionário de Dados](#)

[Vídeo adicional](#)

3.3.6. Normalização do Modelo Lógico

A Normalização é um conceito introduzido em 1970 por Edgard F. Codd e se utiliza de um processo matemático formal com fundamento na teoria dos conjuntos para melhorar o banco de dados e torná-lo mais consistente e íntegro.

O processo de Normalização aplica uma série de regras sobre as tabelas de um banco de dados para verificar se estas foram corretamente projetadas.

Os objetivos principais da Normalização de tabelas são os seguintes:

- Garantir a integridade dos dados, evitando que informações sem sentido sejam inseridas.
- Organizar e dividir as tabelas da forma mais eficiente possível, diminuindo a redundância e permitindo a evolução do banco de dados.

Normalmente, durante a aplicação das regras de normalização, algumas tabelas acabam sendo divididas em duas ou mais tabelas. Este processo colabora significativamente para a estabilidade do modelo de dados e reduz consideravelmente as necessidades de manutenção.

Anomalias de Projeto de Banco de Dados

Quando criamos um modelo de banco de dados, muitas vezes é necessário um refinamento adicional, para que sejam diminuídas e corrigidas situações de:

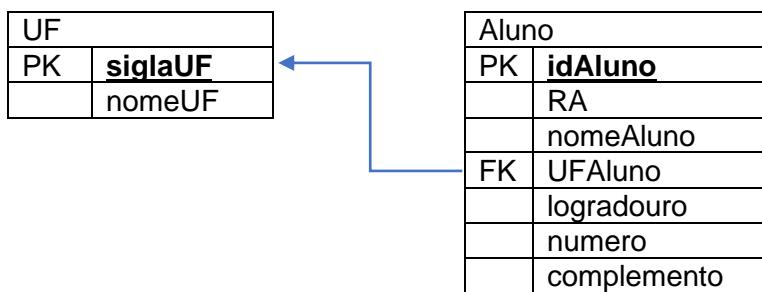
- Redundância ou excesso de informação armazenada em uma mesma tabela
- Unificação de nomenclatura de valores
- Atributos multivvalorados
- Viabilização física dos relacionamentos muitos-para-muitos

Essas situações decorrem do que se chama anomalias, que são derivadas de dependências parciais e transitivas.

Elas aparecem no uso do banco de dados, geralmente nas operações de inserção, exclusão e modificação de dados, e tornam o acesso ao banco de dados menos eficiente e, até mesmo, sem confiabilidade. Temos que procurar evitá-las ao máximo e, para isso, usamos o processo de Normalização. Antes de aprofundarmos nosso estudo da Normalização, vamos explorar um pouco das anomalias.

Anomalia de Inclusão:

Essa situação ocorre quando existe uma dependência funcional entre entidades. Por exemplo, leve em consideração as entidades Aluno e Unidade da Federação (UF): não se poderá incluir (cadastrar) um aluno na tabela Aluno sem que a unidade da federação em que ele mora já esteja cadastrada na tabela UF. Isso ocorre porque existe uma dependência funcional entre essas tabelas. UFAluno seria uma chave estrangeira obrigatória (not null) da tabela Aluno referenciando a chave primária SiglaUF da tabela UF, como vemos na figura abaixo:



Esse mesmo tratamento deve ser feito, por exemplo, para as tabelas Filme e Categoria, como vimos anteriormente na seção sobre Dicionário de Dados. Um filme, para ser incluído, precisa ser de uma categoria já cadastrada na tabela Categoria.

Observe que a tabela UF é considerada uma **entidade forte** em relação à tabela Aluno que seria a **entidade fraca** (ver explicação sobre entidades fortes e fracas [aqui](#)).

Esse cuidado deve ser tomado para manter a Integridade Referencial no Banco de Dados.

Anomalia de Exclusão:

Essa anomalia ocorreria se, ao excluirmos um registro de uma **entidade forte**, deixássemos registros **órfãos** na **entidade fraca**.

Ocorre também quando existe uma dependência funcional entre entidades. Por exemplo, nas entidades Aluno e Unidade da Federação (UF): acima não se poderá excluir (remover) uma unidade da federação da tabela UF quando já existirem, na tabela Aluno, registros cujo campo UFAluno sejam iguais ao valor da chave primária SiglaUF que se deseja excluir.

Se isso ocorrer, alguns registros de alunos poderão referenciar um valor de SiglaUF que não mais existe, e isso quebraria a integridade referencial das duas tabelas.

Uma solução que o sistema de banco de dados poderia ser configurado para fazer seria também remover todos os alunos nessa situação (mesma UF que a que se deseja excluir). Isso se chama exclusão em cascata. Talvez nem sempre seja a solução adequada.

Caso não se configure exclusão em cascata, o sistema de banco de dados impede que essa exclusão seja realizada, abortando-a e impedindo, assim, a quebra da integridade referencial.

Anomalia de Modificação:

Essa anomalia ocorreria se, ao modificarmos a chave primária de um registro de uma **entidade forte**, deixássemos de modificar os registros **correspondentes** na **entidade fraca**. Isso quebraria a integridade referencial entre as tabelas.

Ocorre também quando existe uma dependência funcional entre entidades. Por exemplo, nas entidades Aluno e Unidade da Federação (UF):acima não se poderá modificar a SiglaUF 'MT' para um novo valor, por exemplo 'MO', sem que isso acarrete na alteração de todas as UFAlunos que atualmente valem 'MT' para que passem a valer MO.

Outro problema que poderia ocorrer na alteração da chave primária da entidade forte seria alterar para um valor de chave já existente nessa tabela, pois chaves primárias não podem ser repetidas.

Normalização

A Normalização é um processo de refinamento das relações (tabelas) do Modelo Entidade-Relacionamento para garantir que sejam bem formadas, atendendo aos objetivos acima definidos.

Esse processo foca na busca de **anomalias** nas relações para decompô-las em relações menores e melhor estruturadas, de forma que, quando se executarem operações de inclusão, exclusão e/ou modificação de registros, não ocorram anomalias entre as tabelas relacionadas.

Em geral, durante o processo de Normalização, atributos de uma relação podem ser transferidos para outras relações, podem-se criar chaves estrangeiras, podem-se criar relações para armazenar atributos multivvalorados, dentre outras atividades. Tudo isso dentro de um processo de análise e refinamento sucessivo do Modelo Entidade-Relacionamento visando criar um conjunto de tabelas mais coeso, robusto e consistente.

O Modelo que foi feito até este momento poderá apresentar falhas ao ser transformado em um banco de dados físico. Por isso a Normalização é bastante importante, porque ela refina o modelo e o torna mais correto.

Formas Normais são resultados de operações de Normalização, aplicadas sobre as tabelas e atributos de um banco de dados, para garantir os objetivos acima. São seis as Formas Normais mais utilizadas:

- 1FN – Primeira Forma Normal
- 2FN – Segunda Forma Normal
- 3FN – Terceira Forma Normal
- FNBC – Forma Normal de Boyce e Codd
- 4FN – Quarta Forma Normal
- 5FN – Quinta Forma Normal

As três primeiras Formas Normais atendem à maioria dos casos de Normalização.

Uma forma normal engloba todas as anteriores cumulativamente, isto é, para que uma tabela esteja na 2FN, ela obrigatoriamente deve estar na 1FN e assim por diante.

O ideal é que o modelo de banco de dados esteja, ao menos, na Terceira Forma Normal ou, ainda mais desejável, pelo menos na Forma Normal de Boyce e Codd. Normalizar um banco de dados até somente a Primeira ou Segunda Formas Normais não é uma boa prática, pois ainda permite que o banco de dados apresente falhas estruturais, que o tornam ineficiente e inconsistente.

A Primeira e a Segunda Formas Normais são necessárias para se chegar à Terceira Forma Normal ou à FNBC.

Da mesma maneira, o processo de normalização até a 3FN ou FNBC deve ser aplicado a todas as tabelas, não apenas a algumas.

Antes de passarmos para a Primeira Forma Normal, é importante estudarmos um assunto correlato, que é a Dependência Funcional.

Dependência Funcional

A Dependência Funcional (DF) estabelece uma associação entre atributos de uma tabela. Isso ocorre quando um atributo depende de outro atributo, de tal forma que o valor de um dos atributos determina o valor do outro.

Imagine uma entidade E, e dois atributos X e Y pertencentes a E:

Dizemos que Y é funcionalmente dependente de X se e somente se cada valor de X tiver associado a ele exatamente um único valor de Y e representamos essa associação como

$$X \rightarrow Y$$

que é lido como

“X determina funcionalmente Y” ou “Y é dependente funcionalmente de X”.

Nesse caso, dizemos que X é o determinante e Y é o dependente.

Na tabela ao lado, temos que cidade → estado ou estado é funcionalmente dependente de cidade ou, ainda, **cidade determina estado**. Isso significa que não poderá haver duas cidades de Valinhos, cada uma em um estado diferente.

Local	cidade	estado
	Campinas	São Paulo
	Alcântara	Maranhão
	Valinhos	São Paulo

Outro exemplo: imagine uma tabela de catálogo de disciplinas, descrita como abaixo:

Catalogo	codDisciplina	cargaHoraria	preRequisito
	DS101	120	
	DS102	45	
	DS103	45	
	DS201	90	DS101
	DS202	45	DS101 & DS102

A carga horária e o pré-requisito dependem do código da disciplina considerado. Portanto, codDisciplina é o atributo Determinante e cargaHoraria e preRequisito são atributos Dependentes.

Portanto, quando temos uma chave primária, esta determina funcionalmente todos os outros atributos não-chave de uma linha de dados da relação.

$$\text{codDisciplina} \rightarrow \text{cargaHoraria}$$

$$\text{codDisciplina} \rightarrow \text{preRequisito}$$

Quando temos chaves primárias compostas, um atributo não-chave que dependa dessa chave primária como um todo, e não somente de um ou outro campo que a forma, é dito como possuindo **Dependência Funcional Total**.

Como exemplo, considere a tabela abaixo, chamada **Desempenho**: A chave primária dessa tabela é composta pela composição dos campos RA, codDisciplina e semestre, nessa ordem. Os atributos nota, frequência e situação não dependem somente de um campo da chave primária, e sim da junção entre os três campos componentes da chave. Em outras palavras, por exemplo, através da nota 8,8 não se pode determinar qual aluno a tirou, pois há repetições dessa nota em mais de uma linha de dados.

Para identificar exatamente em que circunstâncias essa nota apareceu, precisamos identificar o trio de valores da chave primária associada a cada ocorrência desse valor de nota:

Desempenho

RA	codDisciplina	semestre	nota	frequencia	situacao	nomeDisciplina
20205	DS101	1	8,8	89,5	Aprovado	Técnicas Program 1
20205	DS102	1	7,5	57,0	Retido	Bancos de Dados I
20205	DS103	1	9,2	98,0	Aprovado	Desenv Internet I
20210	DS101	1	7,7	100,0	Aprovado	Técnicas Program. I
20210	DS102	1	8,8	78,0	Aprovado	Bancos de Dados 1
20210	DS201	2	7,2	91,0	Aprovado	Técnicas Program. II
20210	DS202	3	8,3	75,0	Aprovado	Bancos de Dados 2
20211	DS304	3	3,3	33	Retido	Métodos Ágeis Desenv

Essa nota apareceu em mais de uma linha, no semestre 1 e com situação Aprovado. Como sabemos exatamente em que circunstâncias ela ocorreu? Pela composição das chaves primárias. Para diferenciarmos e identificarmos cada ocorrência dessa nota de forma unívoca (sem dubiedade), temos que identificar com precisão as chaves primárias e, para isso, temos que compor entre campos para que não haja repetição:

(20205, DS101, 1, 8.8 89,5, Aprovado)
 (20210, DS102, 1, 8.8 78,0, Aprovado)

Desempenho do aluno em
uma disciplina e um semestre

Assim, podemos dizer que

{RA, codDisciplina, semestre} → nota.
 {RA, codDisciplina, semestre} → frequencia
 {RA, codDisciplina, semestre} → situacao

Assim, uma nota específica depende, para ocorrer, de um RA, um código de disciplina e um semestre. Esses três campos (RA, codDisciplina e semestre), tratados em conjunto, formam a chave primária dessa tabela

Por depender de todos os 3 campos componentes da chave, essa dependência funcional é chamada de **Dependência Funcional Total**.

Se um atributo depende de parte dos campos de uma chave composta, então a dependência é chamada de **Dependência Funcional Parcial**.

Observe o campo nomeDisciplina da relação Desempenho:

RA	codDisciplina	semestre	nota	frequencia	situacao	nomeDisciplina
20205	DS101	1	8,8	89,5	Aprovado	Técnicas Program I
20205	DS102	1	7,5	57,0	Retido	Bancos de Dados I
20205	DS103	1	9,2	98,0	Aprovado	Desenv Internet I
20210	DS101	1	7,7	100,0	Aprovado	Técnicas Program. I
20210	DS102	1	8,8	78,0	Aprovado	Bancos de Dados 1
20210	DS201	2	7,2	91,0	Aprovado	Técnicas Program. II
20210	DS202	3	8,3	75,0	Aprovado	Bancos de Dados 2
20211	DS304	3	3,3	33	Retido	Métodos Ágeis Desenv

O campo nomeDisciplina depende de quem? Ele não depende do RA nem do semestre. Depende apenas do código da disciplina. Esse campo é parte da chave primária e, assim, temos uma **dependência parcial**.

Observe, no entanto, que nesse caso podemos ter redundância de informação, pois o mesmo nome de disciplina poderá aparecer várias vezes, e ele depende diretamente de um dos campos da chave. Poderia também ocorrer de o usuário digitar, em um dos registros de codDisciplina igual a 'DS101',

um nome de disciplina como ‘Técnicas de Programação I’ e, em outro registro de mesmo código de disciplina, ‘Técnicas Program I’ Em outras palavras, essa abordagem abre a porta para valores diferentes de nome de disciplina para cada ocorrência de um mesmo código de disciplina e, quando isso não ocorre, permite redundância de informação, quando o próprio código já identificaria bem uma disciplina.

Dependência Funcional Transitiva

Uma outra possibilidade de dependência funcional seria a **Dependência Funcional Transitiva**. Ela ocorre quando um campo da relação não depende nem totalmente nem parcialmente da chave primária, mas sim depende de algum outro campo não-chave.

Usemos novamente a tabela de Desempenho, mas imagine que há outra tabela associada, a de Catálogo de Disciplinas, sendo que o codDisciplina de Desempenho é uma chave estrangeira referenciando a chave primária da tabela Catalogo:

Catalogo

codDisciplina	cargaHoraria	preRequisito
DS101	120	
DS102	45	
DS103	45	
DS201	90	DS101
DS202	45	DS101 & DS102
DS304	45	DS201 & DS202

Desempenho

RA	codDisciplina	semestre	nota	frequencia	situacao	nomeDisciplina
20205	DS101	1	8,8	89,5	Aprovado	Técnicas Program I
20205	DS102	1	7,5	57,0	Retido	Bancos de Dados I
20205	DS103	1	9,2	98,0	Aprovado	Desenv Internet I
20210	DS101	1	7,7	100,0	Aprovado	Técnicas Program. I
20210	DS102	1	8,8	78,0	Aprovado	Bancos de Dados 1
20210	DS201	2	7,2	91,0	Aprovado	Técnicas Program. II
20210	DS202	3	8,3	75,0	Aprovado	Bancos de Dados 2
20211	DS304	3	3,3	33	Retido	Métodos Ágeis Desenv

O campo **situacao** depende de quem? Ele não depende do RA, nem do código da disciplina, nem do semestre, que são os atributos da chave primária composta. **situacao** depende simultaneamente dos campos nota e frequência, que não são parte da chave primária. Esse campo é parte da chave primária e, assim, temos uma **dependência funcional transitiva**. Isso leva à situação em que o campo situacao não deveria estar descrito nominalmente na tabela Desempenho, mas poderia ser uma chave estrangeira que referenciasse uma tabela com as possíveis situações dos alunos. Além disso, por ser uma dependência funcional parcial, **nomeDisciplina** também não deveria estar na tabela Desempenho, e sim ser colocado na tabela Catalogo. Essas correções de anomalias vemos na figura seguinte:

Catalogo

codDisciplina	nomeDisciplina	cargaHoraria	preRequisito
DS101	Técnicas de Programação I	120	
DS102	Bancos de Dados I	45	
DS103	Desenvolvimento para Internet I	45	
DS201	Técnicas de Programação II	90	DS101
DS202	Bancos de Dados II	45	DS101 & DS102
DS304	Métodos Ágeis de Desenvolvimento	45	DS201 & DS202

Desempenho

RA	codDisciplina	semestre	nota	frequencia	idSituacao
20205	DS101	1	8,8	89,5	4
20205	DS102	1	7,5	57,0	5
20205	DS103	1	9,2	98,0	4
20210	DS101	1	7,7	100,0	4
20210	DS102	1	8,8	78,0	4
20210	DS201	2	7,2	91,0	4
20210	DS202	3	8,3	75,0	4
20211	DS304	3	3,3	33	5

Situacao

idSituacao	descricao
1	Matriculado
2	Desistente
3	Trancado
4	Aprovado
5	Retido
6	Recuperação
7	Conselho

Observe que, agora, temos uma única ocorrência de cada nome de disciplina e que, por isso mesmo, não teremos mais duas ou mais maneiras diferentes de fornecer um valor para esse campo, pois ele foi unificado. Essa mudança entre os campos manteve a Integridade Referencial e, também, permitiu que diminuíssemos a redundância de informações e o espaço de armazenamento ocupado no Banco de Dados pelo campo nomeDisciplina.

Usando esse raciocínio, também evitamos redundâncias e formas distintas de escrever o campo situação, ao tratar da dependência funcional transitiva desse campo.

Outro exemplo de transitividade ocorre quando um atributo X determina um atributo Y e, se Y determina o atributo Z, podemos dizer que X determina Z de forma transitiva, ou seja, existe uma dependência funcional transitiva de X para Z. Por exemplo:

cidade → estado e estado → pais portanto cidade → pais

Local	cidade	estado	Pais
	Campinas	São Paulo	Brasil
	Alcântara	Maranhão	Brasil
	Valinhos	São Paulo	Brasil
	Miami	Florida	EUA
	Mortágua	Beira Alta	Portugal
	Charlotte	Carolina do Norte	EUA
	Faro	Algarve	Portugal
	Portimão	Algarve	Portugal

Dependência Funcional Irreduzível à Esquerda

O lado esquerdo de uma dependência funcional é irreduzível quando o determinante está em sua forma mínima, isto é, quando não é possível reduzir a quantidade de atributos determinantes sem perder a dependência funcional.

$\{\text{cidade}, \text{estado}\} \rightarrow \text{país}$ (não está na forma irreduzível à esquerda, pois podemos ter somente o estado como determinante)

$\text{estado} \rightarrow \text{país}$ (está na forma irreduzível à esquerda)

<u>cidade</u>	<u>estado</u>	pais
Campinas	São Paulo	Brasil
Atibaia	São Paulo	Brasil
Miami	Florida	EUA
Faro	Algarve	Portugal
Charlotte	Carolina do Norte	EUA

Não irreduzível à esquerda

<u>estado</u>	pais
São Paulo	Brasil
Maranhão	Brasil
Florida	EUA
Beira Alta	Portugal
Algarve	Portugal

Irreduzível à esquerda

Atenção: Nem sempre estar na forma irreduzível à esquerda significa possuir um determinante com apenas uma coluna.

Dependência Funcional Multivalorada

Ocorre quando o valor de um atributo determina **um conjunto** de valores de outro atributo. É uma extensão da Dependência Funcional. É representado como:

$X \rightarrow \rightarrow Y$ e se lê X multidermina Y ou Y é multidependente de X

Imagine a situação abaixo:

Um CPF identifica univocamente um colaborador de uma empresa. No entanto, esse mesmo colaborador poderá ter zero ou mais dependentes. Nesse caso, o mesmo CPF associará esses dependentes ao colaborador, como vemos na próxima tabela:

Colaborador

<u>CPF</u>	nomeColaborador	dependente	dataNasc
173.973.805-90	Antônio da Costa Santos	Maria Célia Gouveia Santos	20/08/2008
		Mário Célio Gouveia Santos	20/08/2008
		Tiago Gouveia Santos	05/03/2015
201.294.157-73	Adélia Prado	Aparecida Celi Prado	31/05/2009
089.199.837-21	Onofre Bagaglio	null	null
745.205.805-05	Quintino Bocaiúva	Silmara Bocaiúva	24/09/2017
801.327.854-32		Jefferson Bocaiúva	29/02/2020
	Antônio da Costa Santos	null	Null

Embora na modelagem possamos pensar em algo assim, não é possível implementar fisicamente esse tipo de atributo multivalorado, ainda mais porque um colaborador pode ter N dependentes e N não é igual para cada instância dessa entidade. Também prever, por exemplo, até 5 dependentes por colaborador poderá parecer válido, mas e se ocorrer de um único colaborador possuir 7 dependentes? Nesse caso nem todos os dependentes poderiam ser armazenados e isso faria com que o banco de dados não fosse confiável.

Essa situação poderia ocorrer não apenas com dependentes, mas também com outros atributos. Por exemplo, um colaborador pode ter mais de um telefone.

Durante o processo de normalização aprenderemos a resolver esse tipo de situação.

[Vídeo adicional](#)

Primeira Forma Normal

A regra da Primeira Forma Normal busca resolver o problema dos atributos multivalorados, que é exatamente o problema que citamos na seção anterior.

Ela diz que: o domínio de um atributo deve incluir apenas valores atômicos (simples, indivisíveis) e o valor de qualquer atributo de uma tupla (linha de dados/registro da tabela) deve ser o único valor do domínio desse atributo.

Valores atômicos são aqueles que representam o nível mais baixo de detalhamento, não podem ser subdivididos em outros atributos menores. Atributos não atômicos podem ser subdivididos. Por exemplo:

- Designação completa de um nome de curso pode ser subdividida em:
 - nome do curso
 - turno (manhã, tarde e noite)
 - tipo de curso (fundamental, médio, técnico, superior, etc).
- Nome de uma pessoa pode ser subdividido em:
 - prenome
 - sobrenome.
- Endereço pode ser subdividido em
 - tipo de logradouro
 - logradouro
 - número
 - complemento
 - bairro
 - CEP.

Aplicar o processo da primeira forma normal aos atributos de uma tabela significa que não mais teremos, na tabela, dados multivalorados, como os atributos (campos) **dependente** e **dataNasc** da tabela **Colaborador**, vista acima.

Observe que, nessa tabela, os campos **dependente** e **dataNasc** praticamente formam uma tabela dentro da outra; isso se chama **relação aninhada**. Remover esse aninhamento não significa deixar de ter os dados, mas sim reorganizá-los em outra tabela, para atender aos requisitos da Primeira Forma Normal.

Uma tabela estará na Primeira Forma Normal quando:

- Somente possui valores atômicos
- Há apenas um único dado por coluna nas linhas
- Existe uma chave primária
- Não há atributos multivalorados ou relações aninhadas
- Não há atributos subdivisíveis.

Vamos colocar a tabela Colaborador na Primeira Forma Normal:

CPF	nomeColaborador	dependente	dataNasc
173.973.805-90	Antônio da Costa Santos	Maria Célia Gouveia Santos	20/08/2008
		Mário Célio Gouveia Santos	20/08/2008
		Tiago Gouveia Santos	05/03/2015
201.294.157-73	Adélia Prado	Aparecida Celi Prado	31/05/2009
089.199.837-21	Onofre Bagaglio	null	null
745.205.805-05	Quintino Bocaiúva	Silmara Bocaiúva	24/09/2017
		Jefferson Bocaiúva	29/02/2020
801.327.854-32	Antônio da Costa Santos	null	Null

PK, Atômico

Não-atômico

Multivalorado, não-atômico

multivalorado

Devemos separar os componentes do nomeColaborador em, pelo menos, dois novos campos: prenome e sobrenome e criar uma nova tabela de Dependentes, relacionada à tabela Colaborador:

Colaborador		
CPF	prenome	sobrenome
173.973.805-90	Antônio	da Costa Santos
201.294.157-73	Adélia	Prado
089.199.837-21	Onofre	Bagaglio
745.205.805-05	Quintino	Bocaiúva
801.327.854-32	Antônio	da Costa Santos

PK, Atômico Atômico Atômico

A tabela Colaborador, agora, está na Primeira Forma Normal (1FN): tem uma chave primária, seus atributos são todos atômicos.

Dependente

CPF	depPrenome	depSobrenome	dataNasc
173.973.805-90	Maria Célia	Gouveia Santos	20/08/2008
173.973.805-90	Mário Célio	Gouveia Santos	20/08/2008
173.973.805-90	Tiago	Gouveia Santos	05/03/2015
201.294.157-73	Aparecida Celi	Prado	31/05/2009
745.205.805-05	Silmara	Bocaiúva	24/09/2017
745.205.805-05	Jefferson	Bocaiúva	29/02/2020

Note que criamos uma nova tabela, **Dependente**. Nessa tabela colocamos os dados dos dependentes de cada colaborador e, para realizar a associação entre essa tabela e a Colaborador, usamos a chave estrangeira CPF. Claramente CPF não pode ser uma chave primária nessa tabela, pois é necessária a repetição dos CPFs para identificar o responsável por cada dependente, de forma isolada e não mais grupalmente.

Mas a tabela Dependente não está na 1FN pois não possui chave primária. Uma solução para isso seria criar uma [chave primária surrogada ou incremental](#). Então, a tabela passaria a ficar como abaixo:

Dependente

idDependente	CPF	depPrenome	depSobrenome	dataNasc
1	173.973.805-90	Maria Célia	Gouveia Santos	20/08/2008
2	173.973.805-90	Mário Célio	Gouveia Santos	20/08/2008
3	173.973.805-90	Tiago	Gouveia Santos	05/03/2015
4	201.294.157-73	Aparecida Celi	Prado	31/05/2009
5	745.205.805-05	Silmara	Bocaiúva	24/09/2017
6	745.205.805-05	Jefferson	Bocaiúva	29/02/2020

PK, Atômica FK, Atômica Atômica Atômica Atômica

Se houvesse um endereço do colaborador, também poderíamos dividí-lo em partes, como as divisões citadas anteriormente.

Você poderia pensar em dividir a data de nascimento em dia, mês e ano mas, na verdade, isso não é necessário porque o sistema de banco de dados armazena a data como um valor único, do domínio (tipo) Datetime. Portanto, ele é um tipo atômico. Caso você precise saber o dia ou mês ou ano de nascimento separadamente, existem funções embutidas no sistema de banco de dados que permitem obter essa informação parcial sobre a data.

Temos outro exemplo, a seguir. Uma empresa de engenharia organiza seus trabalhos com base em Projetos, que contam com colaboradores para neles atuar. Abaixo temos uma planilha que demonstra a alocação de cada colaborador em um ou mais projetos:

Arquivo Página Inicial Inserir Layout da Página Fórmulas Dados Revisão Exibir Ajuda Diga-me Compartilhar

F9 : X ✓ fx

Projeto						
1						
2	Número do Projeto	00001				
3	Nome do Projeto	Instalação Placas Solares				
4	Local do Projeto	Campinas				
5	Funcionário	Cargo	Valor/Hora			
6	Igor Camargo	Engenheiro Eletricista	80,00			
7	Isabel Carmo	Projetista CAD	45,00			
8	Guilherme Moreira	Instalador	35,00			

Projeto						
1						
2	Número do Projeto	00002				
3	Nome do Projeto	Infraestrutura Home Office				
4	Local do Projeto	São Paulo				
5	Funcionário	Cargo	Valor/Hora			
6	Isabel Carmo	Projetista CAD	45,00			
7	Guilherme Moreira	Instalador	50,00			
8	Luis Felipe	Analista de Sistemas	80,00			

Projetos + 100%

Essa mesma planilha foi reorganizada para fornecer uma visão mais abrangente para o gestor dos projetos:

Arquivo Página Inicial Inserir Layout da Página Fórmulas Dados Revisão Exibir Ajuda Diga-me Compartilhar

D6 : X ✓ fx Guilherme Moreira

A	B	C	D	E	F	
1	Número do Projeto	Nome do Projeto	Local do Projeto	Funcionário	Cargo	Valor/Hora
2	00001	Instalação Placas Solares	Campinas	Igor Camargo	Engenheiro Eletricista	80,00
3				Isabel Carmo	Projetista CAD	45,00
4				Guilherme Moreira	Instalador	35,00
5	00002	Infraestrutura Home Office	São Paulo	Isabel Carmo	Projetista C.A.D.	45,00
6				Guilherme Moreira	Instalador	50,00
7				Luis Felipe	Analista de Sistemas	80,00

Projetos Tabela + 100%

No entanto, a empresa decidiu adquirir um sistema de banco de dados para gerenciar seus projetos de forma mais eficiente, confiável e rápida.

O técnico que produziu o sistema de banco de dados baseou-se na planilha acima para modelar os dados necessários, e decidiu criar tabelas normalizadas em 1FN, primeiramente. Observou o seguinte:

- Não há um identificador único para cada funcionário
- É possível digitar os dados sem nenhuma consistência (por exemplo, um mesmo cargo escrito de duas formas diferentes: Projetista CAD e Projetista C.A.D.)
- O valor/hora não depende apenas do funcionário, mas também do projeto em que este atua.
- Há tabelas aninhadas
- Os campos funcionário, cargo e valor/hora são multivvalorados

O técnico primeiramente separou a tabela em duas outras, Projeto e Local, como vemos abaixo:

Projeto		
idProjeto	nome	idLocal
1	Instalação Placas Solares	1

Local	
idLocal	local
1	Campinas

→

2	Infraestrutura Office	Home	3	
PK, atômica, auto-incremento	Atômica	FK, atômica	PK, atômica, auto-incremento	Atômica

E, para manter os dados na 1FN, criou tabelas que relacionam os funcionários, seus cargos e valor/hora com cada projeto:

Funcionario		
<u>idFuncionario</u>	nomeFuncionario	idCargo
1	Igor Camargo	3
2	Isabel Carmo	2
3	Guilherme Moreira	4
4	Luís Felipe Assis	1

PK, auto-inc Atômico

FK, atôm

ProjetoFuncionario		
<u>idProjeto</u>	<u>idFuncionario</u>	valorHora
1	1	80,00
1	2	45,00
1	3	35,00
2	2	45,00
2	3	50,00
2	4	80,00

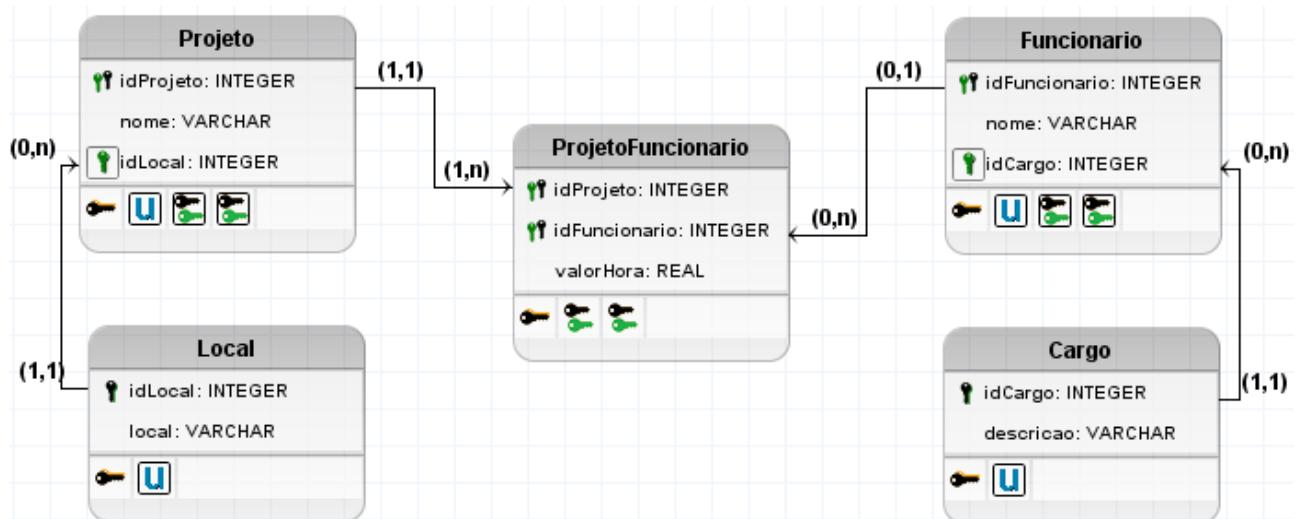
PK composta por:
(idProjeto, idFuncionario)

Atômico

Cargo	
<u>idCargo</u>	descricao
1	Analista de Sistemas
2	Projetista CAD
3	Engenheiro Eletricista
4	Instalador

PK, auto-inc atômico

Abaixo temos um diagrama mais detalhado que mostra os relacionamentos entre essas tabelas normalizadas:



Segunda Forma Normal

Uma tabela está na 2FN (Segunda Forma Normal) quando, além de estar na Primeira Forma Normal, não contém dependências parciais.

Uma dependência funcional parcial ocorre quando uma coluna depende apenas de uma parte da Chave Primária COMPOSTA, como vimos anteriormente.

Portanto, para uma tabela estar na Segunda Forma Normal, cada um dos seus atributos não-chave dessa tabela terá de ser total e funcionalmente dependente somente da chave primária da tabela, se ela for composta, ou seja, todos os campos não-chave deverão depender da chave primária em sua totalidade.

Se a chave primária não for composta e a tabela já estiver na 1FN, então ela estará também na 2FN.

Voltemos a observar uma das versões das nossas tabelas de estudo anterior, Catalogo e Desempenho:

Catalogo	codDisciplina	cargaHoraria	preRequisito
DS101	120		
DS102	45		
DS103	45		
DS201	90	DS101	
DS202	45	DS101 & DS102	
DS304	45	DS201 & DS202	

Desempenho

RA	codDisciplina	semestre	nota	frequencia	situacao	nomeDisciplina
20205	DS101	1	8,8	89,5	Aprovado	Técnicas Program I
20205	DS102	1	7,5	57,0	Retido	Bancos de Dados I
20205	DS103	1	9,2	98,0	Aprovado	Desenv Internet I
20210	DS101	1	7,7	100,0	Aprovado	Técnicas Program. I
20210	DS102	1	8,8	78,0	Aprovado	Bancos de Dados 1
20210	DS201	2	7,2	91,0	Aprovado	Técnicas Program. II
20210	DS202	3	8,3	75,0	Aprovado	Bancos de Dados 2
20211	DS304	3	3,3	33	Retido	Métodos Ágeis Desenv

Na tabela Catalogo, campo cargaHoraria depende total e funcionalmente da chave primária, codDisciplina. O mesmo ocorre para o campo preRequisito, que não depende da carga horária. Portanto, essa tabela está na 1FN e na 2FN.

Já na tabela Desempenho, a chave primária é composta pelos campos {RA, codDisciplina, semestre} e o campo nomeDisciplina depende apenas de parte da chave primária, ou seja, o campo codDisciplina. Essa tabela não está na 2FN. Para deixá-la na 2FN, o campo nomeDisciplina terá de ser removido dessa tabela e acoplado a outra tabela.

No caso, será movido para a tabela Catalogo. Note que já há um relacionamento entre essas duas tabelas, onde codDisciplina de Desempenho é chave estrangeira que referencia a chave primária codDisciplina da tabela Catalogo.

Transferimos o campo nomeDisciplina para a tabela Catalogo, de modo a remover a dependência funcional parcial, e as duas tabelas ficarão assim:

Catalogo

codDisciplina	nomeDisciplina	cargaHoraria	preRequisito
DS101	Técnicas de Programação I	120	
DS102	Bancos de Dados I	45	
DS103	Desenvolvimento para Internet I	45	
DS201	Técnicas de Programação II	90	DS101
DS202	Bancos de Dados II	45	DS101 & DS102
DS304	Métodos Ágeis de Desenvolvimento	45	DS201 & DS202

Desempenho

RA	codDisciplina	semestre	nota	frequencia	situacao
20205	DS101	1	8,8	89,5	Aprovado
20205	DS102	1	7,5	57,0	Retido
20205	DS103	1	9,2	98,0	Aprovado
20210	DS101	1	7,7	100,0	Aprovado
20210	DS102	1	8,8	78,0	Aprovado
20210	DS201	2	7,2	91,0	Aprovado
20210	DS202	3	8,3	75,0	Aprovado
20211	DS304	3	3,3	33,0	Retido

Aluno

RA	Nome	anoIngresso	idCidade	...
20205	Luis Afonso Ferreira	2020	1	...
20210	Célia Regina Duarte	2020	2	...
20211	Clélia Roberta Duarte	2020	2	...

Além das tabelas acima, também deveremos ter a tabela Aluno, cuja chave primária é o RA, e que descreve os dados de um aluno. Desempenho, portanto, possui relacionamentos com Aluno e com Catalogo.

Observe que diminuímos a redundância de informações, pois passamos a ter apenas uma instância de cada nome de disciplina específica.

Se a tabela Catalogo ainda não existisse, ela teria de ser criada.

Deve-se criar uma nova tabela para cada chave PK ou combinação de atributos que forem determinantes em uma dependência funcional e movemos os atributos não-chave dependentes dessa chave primária para essa nova tabela.

Faremos essa análise da dependência de atributos em relação a partes da chave primária para todos os atributos não-chave. Se houver mais um atributo que dependa de um dos atributos componentes da chave primária composta, realizaremos a mesma operação. Se esse atributo depender do mesmo atributo parte da chave que antes, o atributo não-chave será deslocado para a nova tabela. Caso contrário, outra tabela terá de ser criada para armazenar os dados desse atributo não-chave que depende parcialmente da chave primária composta.

[Retorne ao item em que discutimos o banco de dados de Projetos](#) e observe que suas tabelas também já estão na 2FN.

Nesse item observe que, na tabela ProdutoFuncionario, a chave primária é composta pelos campos idProjeto e idFuncionario. O campo valorHora depende dessas duas chaves, e não apenas de uma delas. Por esse motivo, essa tabela também está na 2FN. Se o campo valorHora dependesse apenas do idFuncionario, esse campo deveria ser armazenado na tabelas de funcionários para que o esquema desse banco de dados ficasse na 2FN.

[Vídeo adicional](#)

Terceira Forma Normal

Uma relação está na Terceira Forma Normal (3FN) quando, além de estar na 2FN, também não contém nenhuma dependência funcional transitiva.

A relação não deverá ter nenhum atributo não-chave dependente de outro atributo (ou conjunto de atributos) não-chave. Não pode haver dependência transitiva de um atributo não-chave sobre a chave primária.

Quando isso ocorrer, deve-se decompor a relação e montar uma nova relação que inclua os atributos não chave que possuam dependência transitiva com outros atributos não chave. Considere a relação abaixo:

Local	cidade	estado	Pais
Campinas	São Paulo	Brasil	
Alcântara	Maranhão	Brasil	
Valinhos	São Paulo	Brasil	
Miami	Florida	EUA	
Mortágua	Beira Alta	Portugal	
Charlotte	Carolina do Norte	EUA	
Faro	Algarve	Portugal	
Portimão	Algarve	Portugal	

Nela temos que:

$$\text{cidade} \rightarrow \text{estado} \quad \text{e} \quad \text{estado} \rightarrow \text{pais} \quad \text{portanto} \quad \text{cidade} \rightarrow \text{pais}$$

Podemos colocar essa tabela na 3FN fazendo algumas decomposições, como vemos abaixo:

1. Criar uma chave primária incremental (surrogada) para a tabela Local, e tornar o campo cidade como um atributo normal, não-chave;
2. Criar DUAS novas relações, Estado e País, cada uma com sua respectiva chave primária
3. Reorganizar os dados nas relações, como vemos abaixo:

Local		
<u>idLocal</u>	cidade	<u>idUF</u>
1	Campinas	1
2	Alcântara	2
3	Valinhos	1
4	Miami	3
5	Mortágua	4
6	Charlotte	5
7	Faro	6
8	Portimão	6

DivisaoRegional		
<u>idUF</u>	nomeUF	<u>idPais</u>
1	São Paulo	1
2	Maranhão	1
3	Florida	2
4	Beira Alta	3
5	Carolina do Norte	1
6	Algarve	3

País	
<u>idPais</u>	nomePais
1	Brasil
2	Estados Unidos da América
3	Portugal

Observe que tivemos de criar campos adicionais (idUF, idPais) pois eles são necessários para estabelecer os relacionamentos entre as relações acima. Também diminuiu o número de repetições de nomes de estados e países, nomes esses que ficaram unificados numa única instância de cada um, sem diferenças de formas de escrevê-los.

A invés de chave incremental, poderíamos também usar siglas, como SP, ES, SC e BRA, ARG, AUS, EUA mas o que aconteceria com Austrália e Áustria? Também poderíamos ter casos de siglas de Divisões Regionais idênticas para dois países diferentes, já que há tantos países e cada um tem suas divisões regionais. Por exemplo, BA no Brasil é a sigla de Bahia, mas em Portugal BA se refere à região administrativa da Beira Alta. Por isso que se decidiu usar chaves numéricas incrementais ao invés de siglas.

Se houvesse, por exemplo, uma relação **Funcionario**, ela usaria como chave estrangeira um atributo idCidade, que faria referência ao atributo **idCidade** (PK) da relação **Local**.

Temos outro exemplo abaixo, baseado no vídeo adicional sugerido ao final desta seção:

Venda

notaFiscal	idVendedor	nomeVendedor	idProduto	nomeProduto	qtdeVendida
15326	2	Leila Chaves	132	Bono Doce de Leite	10
15327	6	Ana Veltrame	153	Lasanha 4 Queijos	12
15328	2	Leia Chaves	143	Nesfit Canela Maçã	11
15329	9	Fábio Celta	132	Bono Doce Leite	9
15330	7	Renato Russo	153	Nívea Happy Time	12

O atributo nomeProduto não depende da notaFiscal (PK). O mesmo ocorre com nomeVendedor. Portanto, criamos outras relações para abrigar esses atributos.

Já as chaves estrangeiras (FK) nessa relação, dependem da notaFiscal, pois elas fazem parte de cada operação comercial específica. Isso também acontece com qtdeVendida.

Então, teremos as seguintes relações:

Venda

notaFiscal	idVendedor	idProduto	qtdeVendida
15326	2	132	10
15327	6	153	12
15328	2	143	11
15329	9	132	9
15330	7	153	12

Vendedor

idVendedor	nomeVendedor
2	Leila Chaves
6	Ana Veltrame
7	Renato Russo
9	Fábio Celta

Produto

idProduto	nomeProduto
132	Bono Doce de Leite
143	Nesfit Canela Maçã
153	Nívea Happy Time

Também diminuiu o número de repetições de nomes de produtos e de vendedores, nomes esses que ficaram unificados numa única instância de cada um, sem diferenças nas formas de escrevê-los.

Quando você chega à 3FN, seu modelo de banco de dados está minimamente normalizado e conseguirá lidar com a **maior parte** das necessidades de um sistema de bancos de dados.

Ainda é possível normalizar mais o seu modelo, de forma que ainda veremos alguns conceitos de normalização futuramente, mas, para o momento, é o que precisamos nesta fase de nosso estudo.

Observação Importante

Você pode estar se perguntando: se separamos os dados em tantas relações diferentes, como faremos para juntá-los quando precisarmos deles unidos? Por exemplo, quando precisarmos gerar um relatório de vendas?

Bem, nesses momentos, usaremos comandos de uma linguagem especial de acesso a dados de bancos de dados, e nesses comandos existem maneiras de buscar dados relacionados em diferentes tabelas.

Portanto, não se preocupe com isso agora, pois será muito fácil juntar os dados quando precisarmos.

[Vídeo adicional](#)

Exercícios

Provenientes da apostila Modelagem de Bancos de Dados – prof. Marcos Alexandruk

1. Explique quando uma tabela está em conformidade com cada uma das seguintes Formas Normais:

- a. 1FN (Primeira Forma Normal)
- b. 2FN (Segunda Forma Normal)
- c. 3 FN (Terceira Forma Normal)

2. Aplique as três primeiras Formas Normais à tabela PEDIDOS:

PEDIDOS							
NR_PEDIDO	DATA_PEDIDO	ID_CLIENTE	NOME_CLIENTE	COD_PROD	NOME_PROD	QUANT	VL_UNIT
001	10/01/2011	1003	Ernesto	P-31	Caderno	2	15,00
				P-42	Caneta	1	3,00
				P-67	Lápis	5	1,00
002	11/01/2011	1007	Fabiana	P-42	Caneta	2	3,00
				P-67	Lápis	3	1,00
				P-85	Lapiseira	1	5,00

3. Aplique as três primeiras Formas Normais à tabela de DEPARTAMENTOS:

DEPARTAMENTOS						
COD_DEPT	LOCAL	ID_GERENTE	NOME_GERENTE	TIPO_FONE	COD_AREA	NR_FONE
1011	São Paulo	35215	Geraldo	Residencial	12	5555-1234
				Comercial	11	5555-4321
				Celular	11	5555-9876
1021	Rio de Janeiro	47360	Horacia	Residencial	21	5555-5678
				Comercial	22	5555-3659
				Celular	21	5555-2345

4. Aplique as três primeiras Formas Normais à tabela CURSOS:

CURSOS							
COD_CURSO	NOME_CURSO	COD_TURMA	NR_SALA	COD_DISC	NOME_DISC	ID_PROF	NOME_PROF
1005	TADS	1005_3A3	230	3523	Algoritmos	105	Ildemar
				5282	Banco de Dados	118	Joselia
				8346	Empreendedorismo	126	Kleudir
	FEGAIRC	1005_3B3	231	3523	Algoritmos	133	Lucimar
				5282	Banco de Dados	118	Joselia
				8346	Empreendedorismo	126	Kleudir
1250	TADS	1250_4A1	380	4639	Cálculo	133	Lucimar
				6395	Lógica Digital	142	Marcelo
				9578	Redes de Dados	158	Nilmara
	FEGAIRC	1250_4B1	381	4639	Cálculo	133	Lucimar
				6395	Lógica Digital	165	Osvaldo
				9578	Redes de Dados	158	Nilmara

Quarta Forma Normal

Na grande maioria dos casos, as entidades normalizadas até a 3FN são fáceis de entender, atualizar e de se recuperar dados. Mas, às vezes, podem surgir problemas com relação a algum atributo não chave, que recebe valores múltiplos para um mesmo valor de chave.

Este novo tipo de dependência recebe o nome de **dependência multivalorada**, que poderá existir somente se a entidade possuir no mínimo 3 atributos.

Essa forma normal busca tratar das dependências multivaloradas em relações que estejam na 3FN: repetição de atributos não-chave, gerando redundância desnecessária e uma chave primária composta pelos campos redundantes. Por exemplo, observe a entidade abaixo e um exemplo de relação (tabela) com dados, em que uma música pode ser gravada por vários intérpretes e um intérprete pode gravar em diversas gravadoras. São situações que não estão associadas entre si, descrevem duas situações diferentes que, no processo de normalização até a 3FN, foram mantidas na mesma relação (mesma entidade):

Música (nome, intérprete, gravadora)

nome	intérprete	gravadora
Aquarela do Brasil	Toquinho	Polygram
Aquarela do Brasil	Tim Maia	Som Livre
Aquarela do Brasil	Toquinho	Som Livre
Andança	Beth Carvalho	Som Livre
Andança	Roupa Nova	Polygram
Andança	Beth Carvalho	Polygram

nome	intérprete
Aquarela do Brasil	Toquinho
Aquarela do Brasil	Tim Maia
Andança	Beth Carvalho
Andança	Roupa Nova

nome	gravadora
Aquarela do Brasil	Polygram
Aquarela do Brasil	Som Livre
Andança	Som Livre
Andança	Polygram

MusicInterprete (nome, intérprete)

MusicaGravadora (nome, gravadora)

A quarta forma normal assegura que não existam entidades com atributos não chave (que não fazem parte da chave primária) e que possuam valores múltiplos.

Uma entidade que está na 3FN também está na 4FN se ela não contiver mais do que **um único atributo multivalorado** a respeito da entidade descrita. Esta dependência não é o mesmo que uma associação M:N entre atributos, geralmente descrita desta forma. Vejamos o exemplo abaixo:

codigoFornecedor	codigoPeca	codigoCliente
111	BA3	113
111	C10	113
111	88A	435
111	BA3	537

Apesar de estar na 3FN a entidade não representa um conceito válido no **mundo real**. Ela representa **dois conceitos, dois fatos diferentes**. A entidade acima representa o **fornecimento de uma peça pelo fornecedor e a compra da peça pelo cliente**. Vemos que a peça unifica os dois conceitos.

Para passarmos a entidade acima para a 4FN devemos dividi-la em duas outras entidades, cada uma representando um fato isolado, ou seja, a peça sendo fornecida e a peça sendo comprada. Assim, teremos duas entidades uma para cada fato contendo seus atributos específicos.

PecaFornecedor		PecaCliente	
codigoPeca	codigoFornecedor	codigoPeca	codigoCliente
BA3	111	BA3	113
C10	111	C10	113
88A	111	88A	435
		BA3	537

Quinta Forma Normal

Essa forma normal busca simplificar as relações que estejam na 4FN para relações mais simples. Tomemos como exemplo a relação MusicalInterprete acima. Ela pode ser subdividida em três relações pelo uso de uma chave incremental (surrogada):

Musica(idMusica, nome)		MusicalInterprete(idMusica, idInterprete)		Interprete(idInterprete, nome)	
idMusica	nome	idMusica	idInterprete	idInterprete	nome
1	Aquarela do Brasil	1	1	1	Toquinho
2	Andança	1	2	2	Tim Maia
		2	3	3	Beth Carvalho
		2	4	4	Roupa Nova

Forma Normal de Boyce-Codd

Uma relação está na FNBC se todo **determinante** for uma **chave candidata**.

Originalmente, foi proposta como uma extensão da 3FN, e também é conhecida como 3.5FN, mas é considerada mais rígida que essa. Em outras palavras, toda relação na FNBC está na 3FN, porém uma relação na 3FN não necessariamente estará na FNBC.

FBCN pode ser usada porque a 3FN não tratou satisfatoriamente de casos onde uma relação tem mais de uma chave candidata, estas chaves são compostas e possuem atributos em comum. Vejamos o exemplo abaixo:

Ensino (estudante, disciplina, professor) é uma relação com o seguinte significado:

- Em cada Disciplina, cada Estudante recebe aulas de apenas um Professor;
- Cada Professor leciona somente uma Disciplina;
- Uma Disciplina pode ser lecionada por mais de um Professor.

estudante	disciplina	professor
20699	Bancos de Dados I	Francisco
20699	Técnicas de Programação I	Andréia
20320	Bancos de Dados I	Francisco
20320	Técnicas de Programação I	Márcia

Chaves Candidatas
(estudante, disciplina)
(estudante, professor)

Dependências Funcionais
(estudante, disciplina) → professor
professor → disciplina

Se a chave primária for (estudante, disciplina) a relação estará na 3FN. Se a chave primária for (estudante, professor) a relação também estará na 3FN. Em ambos os casos, a relação não estará na FNBC porque o determinante **professor** não é considerado uma chave candidata.

Então, para obter a FNBC, fazemos as seguintes operações:

1. Identificar as dependências funcionais que violem a BCFN
2. Para cada dependência funcional identificada em 1, criar uma nova relação cuja PK seja igual ao determinante
3. As colunas que tem seus valores determinados em 1, são excluídas da relação original.

Ensino(estudante, professor)

<u>estudante</u>	<u>professor</u>
20699	Francisco
20699	Andréia
20320	Francisco
20320	Márcia

Leciona(professor, disciplina)

<u>professor</u>	<u>disciplina</u>
Francisco	Bancos de Dados I
Andréia	Técnicas de Programação I
Márcia	Técnicas de Programação I

Também se poderia criar tabelas específicas de professores e disciplinas, com chaves surrogadas e relacioná-las, como vemos abaixo:

<u>estudante</u>	<u>professor</u>
20699	1
20699	2
20320	1
20320	3

<u>idProfessor</u>	<u>professor</u>	<u>disciplina</u>
1	Francisco	Bancos de Dados I
2	Andréia	Técnicas de Programação I
3	Márcia	Técnicas de Programação I

Essa normalização só será possível se **cada professor somente lecionar uma disciplina durante a validade desses dados**.

Outro exemplo, numa tabela de empréstimos de livros em uma biblioteca:

Emprestimo(ISBN, codEmprestimo, dataDevolucao, codLeitor)

Temos a dependência abaixo:

{ISBN, codEmprestimo} → dataDevolucao, codLeitor

codEmprestimo é chave candidata (ocorrência única)

Portanto, o Banco de Dados ficaria com as tabelas abaixo:

EmprestimoLivro(ISBN, codEmprestimo)

Emprestimo(codEmprestimo, dataDevolucao, codLeitor)

Resumindo:

- A FNBC é mais rígida que a 3FN, mas, na prática quase todas relações que estão na 3FN também estão na FNBC;
- As relações que estão na 3FN e que têm uma única chave candidata estão automaticamente na FNBC;
- Uma relação que está na 3FN e que não tem chaves candidatas sobrepostas está na FNBC;
- A FNBC não faz referência explícita à 1FN e à 2FN e nem ao conceito de dependência transitiva, quando um campo não-chave depende de algum outro campo não-chave.

3.4. Modelo Físico

Após termos realizado a modelagem do banco de dados, normalizado e criado o dicionário de dados, estamos prontos para partir para o modelo físico.

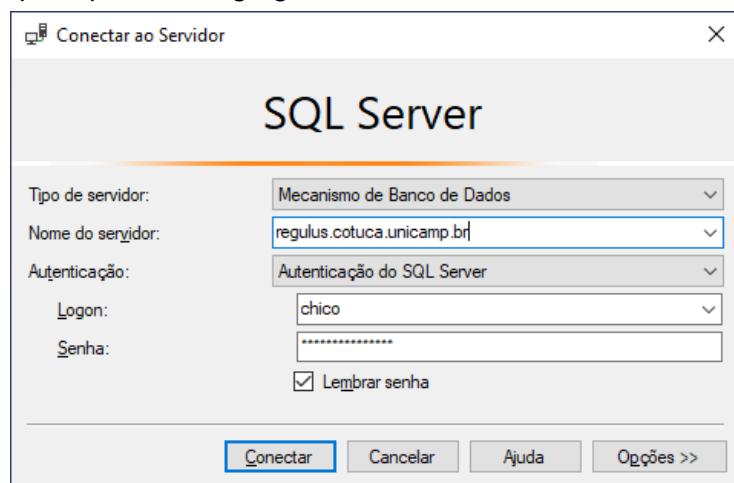
Basicamente, o Dicionário de Dados quando feito a partir do modelo normalizado já será uma aproximação bastante precisa das tabelas e seus relacionamentos criados fisicamente no Servidor de Banco de Dados.

Como estamos já pensando na implantação física das tabelas, atributos e relacionamentos, essa fase depende das características do Servidor de Banco de Dados escolhido. Sabemos que há vários produtos disponíveis no mercado, e que os principais têm algo de confiabilidade e eficiência.

Vamos usar o SQL Server como banco de dados e, portanto, teremos de usar suas configurações e características.

A interface com usuário que utilizaremos é o SQL Server Management Studio (SSMS), cuja tela de logon vem ao lado:

Observe que usarmos o endereço de um servidor da rede da escola para acessar. Para que isso funcione, esse servidor precisará ter sido configurado para acesso remoto e a rede local da escola precisará, também, ter seu firewall configurado para permitir acessos externos à escola para esse servidor.



No link abaixo temos instruções para baixar esse software e instalá-lo.

[Instalação do Sql Server Management Studio](#)

Para você poder acessar o SQL Server da escola remotamente, também terá que executar o aplicativo da conexão VPN para a Unicamp validar seu login (ccAAnnn@unicamp.br) e senha.

 Centro de Computação



Aluno do COTUCA prefixo 'cc' seguido do RA Ex: cc999999@unicamp.br	Professor ou Funcionário usuário Unicamp. Ex: j.silva@unicamp.br
--	--

Segue, abaixo, tutoriais de configuração da VPN para os sistemas operacionais:

 Windows	 Linux	 Mac OS	 iOS (Para iPhone e iPad)
--	--	---	---

 Android
--

O link para informações sobre o acesso da VPN da Unicamp está abaixo:

Para mais informações sobre o serviço acesse [Acesso Remoto Seguro VPN](#).

<https://www.ccuec.unicamp.br/ccuec/euquero/utilizar-o-acesso-remoto-seguro-vpn>

O link do tutorial de instalação para Windows é o que segue:

https://www.ccuec.unicamp.br/ccuec/material_apoio/tutorial_openvpn_windows

O link do tutorial de instalação para Linux vem abaixo:

https://www.ccuec.unicamp.br/ccuec/material_apoio/tutorial_linux_vpn_versoes

O link do tutorial de instalação para MacOS segue abaixo:

https://www.ccuec.unicamp.br/ccuec/material_apoio/mac_os_vpn

Além do SSMS, você também poderá instalar no seu computador pessoal o Sql Server Express, que é um servidor Sql Server reduzido e que é ideal para o aprendizado prático desta disciplina. Assim, terá o servidor diretamente instalado em seu computador e não precisará de Internet para acessar seus bancos de dados.

Downloads do SQL Server | Microsoft

www.microsoft.com/pt-br/sql-server/sql-server-downloads

Acessando o link acima, procure a figura abaixo na página de instalação e terá acesso à versão Express. Se preferir, também poderá instalar a Developer, que é mais completa. Porém a Express tem o necessário para este momento.

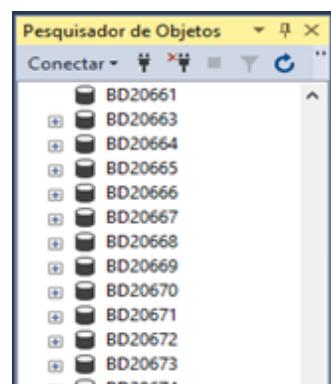


Vídeo adicional sobre instalação do Sql Server Express

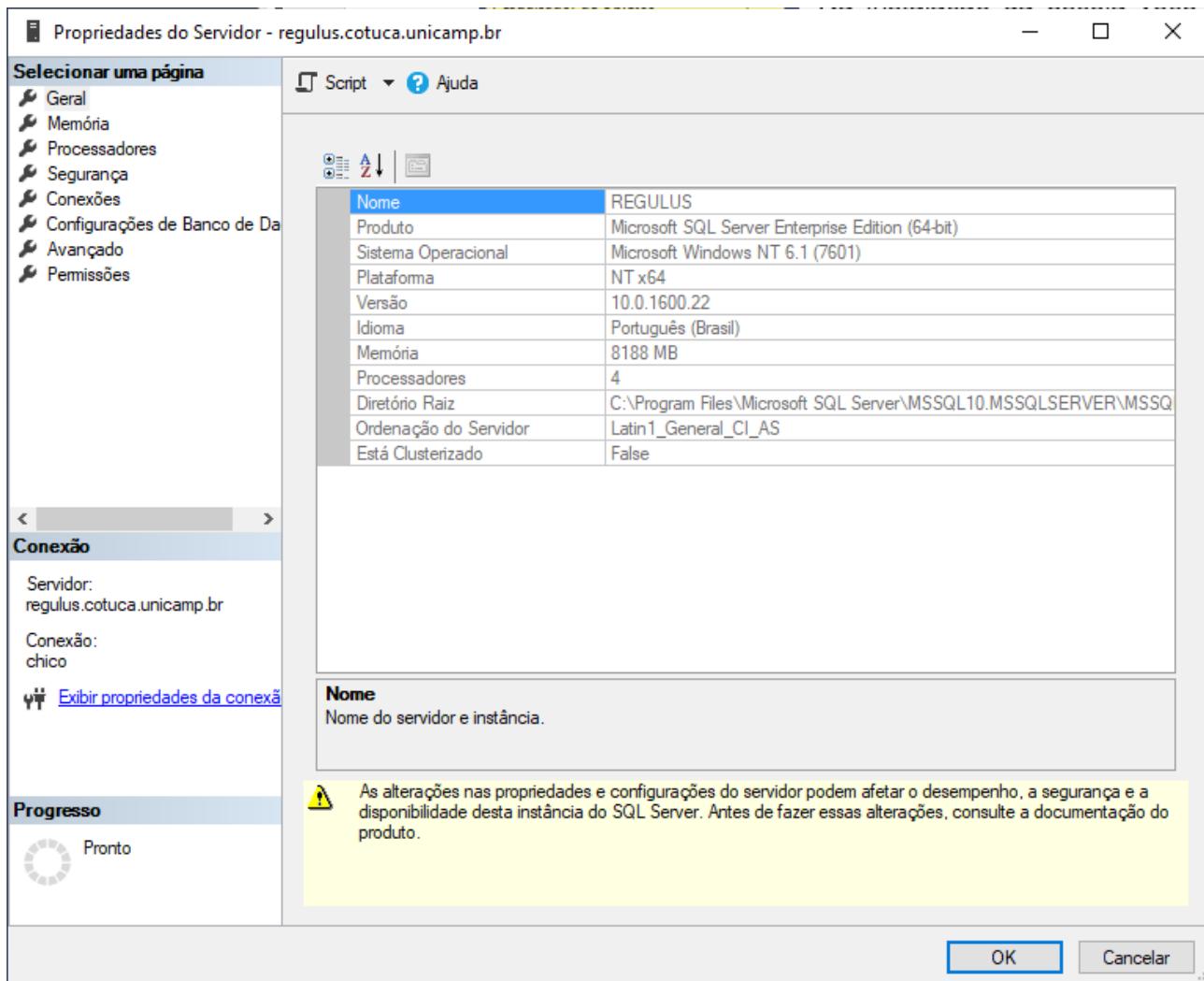
Na instalação da escola você já terá um banco de dados instalado, chamado BD<seuRA> (BD20699, por exemplo) para seu uso.

Caso instale o Sql Server em seu computador pessoal, poderá futuramente copiar as tabelas, relacionamentos, índices e dados para o seu banco de dados na escola, através de mecanismos de backup e restauro.

A figura ao lado mostra parte dos bancos de dados de alunos instalados no servidor de bancos de dados do Cotuca.



Na figura abaixo vemos as características de configuração geral do servidor de banco de dados do Cotuca no momento da digitação deste texto:



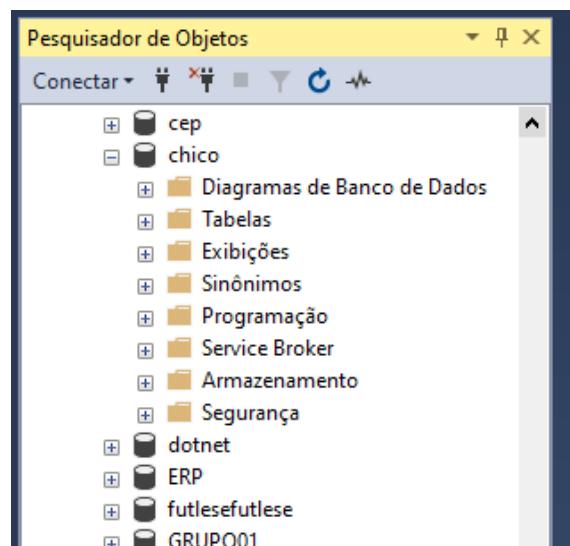
Na figura ao lado vemos as coleções de objetos que formam um banco de dados (chico) no Sql Server. Observe que cada tipo de objeto é organizado em uma pasta específica:

Nosso primeiro passo será, a partir da versão final do Dicionário de Dados, baseado no modelo entidade-relacionamento já normalizado até, pelo menos, a 3FN, vamos criar tabelas usando o SSMS como interface de acesso ao Servidor de Banco de Dados.

Supondo que você já logou no servidor do Cotuca usando a tela de logon do SSMS, procure o seu banco de dados na relação de objetos do Pesquisador de Objetos. Clique no botão [+] para expandir os objetos.

Clique no nome do seu banco de dados e, em seguida, clique no botão [Nova Consulta]. Uma janela de consulta aparecerá.

Elá funciona como um editor de textos, onde você deverá digitar comandos da linguagem SQL para enviar ordens ao Servidor de Banco de Dados. A figura abaixo mostra essa situação, com o comando Create Table que é usado para criar uma tabela de dados:



O botão [Executar] que vemos na figura ao lado envia, para o Servidor de Banco de Dados, o texto do comando digitado no Editor de Consulta.

Esse texto será recebido pelo servidor, analisado e, se o servidor o considerar corretamente escrito, o executará.

No caso ao lado, se o comando já estivesse terminado e correto, seria criada a tabela Aluno dentro da coleção Tabelas que vemos no banco de dados chico da figura acima.

```
CREATE TABLE Aluno
(
    RA char(5) primary key,
    nome varchar(100) not null,
    cpf varchar(15) not null unique,
)
```

SQL ou Structured Query Language é a linguagem que nos permite utilizar um Servidor de Banco de Dados Relacional, como o Sql Server, Oracle ou MySql. Ela é composta por diversos comandos, que permitem trabalhar com a estruturação e configuração física das tabelas, atributos, relacionamentos, índices, triggers, bem como acessar, modificar e remover dados. Além disso, também há comandos de programação para o que chamamos de Funções e Procedimentos Armazenados.

4. SQL – Linguagem de Consulta Estruturada

4.1. Introdução

A linguagem SQL pode ser considerada uma das principais razões para o sucesso dos bancos de dados relacionais. Como ela é relativamente padronizada entre os vários sistemas gerenciadores de bancos de dados, os usuários de um sistema de banco de dados podem ficar menos preocupados sobre migrar suas aplicações de bancos de dados para outros tipos de servidores.

A conversão de um sistema de gerenciamento de banco de dados para outro é relativamente simples, pois a SQL funciona como uma base comum para os diversos fornecedores de SGBDs comerciais. Obviamente que ocorrem diferenças em detalhes entre as soluções dos fornecedores porém, se o desenvolvedor do sistema de banco de dados foi diligente e procurou usar apenas os recursos que são parte do padrão ISO (**International Organization for Standardization**) do SQL, e se o SGBD de origem e o de destino suportam esse padrão, então a conversão entre os dois deve ser suave.

Outra vantagem de SQL ser padronizado é que se poderá reutilizar os comandos escritos para uma aplicação que acessa um SGBD, para acessar um SGBD de outro fornecedor sem grandes alterações de código.

SQL é uma linguagem que foi criada a partir de especificações de [Álgebra Relacional](#), mas não estudaremos esses conceitos matemáticos aqui. A álgebra relacional recebia pouca atenção fora do campo da matemática pura até à publicação em 1970 do [modelo relacional de dados](#) de [E.F. Codd](#). Codd propôs tal álgebra como a base das linguagens de consulta de banco de dados.

SQL foi inicialmente desenvolvida na [IBM](#) por [Donald D. Chamberlin](#) e [Raymond F. Boyce](#), depois que eles souberam do modelo relacional de Codd no princípio da década de 1970s.

Passou a ser considerada um padrão ANSI em 1986 e ISO em 1987. A versão mais recente do padrão é de 2016.

SQL é uma linguagem dividida em vários subconjuntos:

DDL - Linguagem de Definição de Dados

[DDL](#) (Data Definition Language - Linguagem de Definição de Dados) permite ao desenvolvedor definir tabelas novas e elementos associados. A maioria dos bancos de dados de SQL comerciais tem extensões proprietárias no DDL.

Os comandos básicos da DDL são poucos:

- CREATE: cria um objeto (uma [Tabela](#), por exemplo) dentro da base de dados.
- DROP: apaga um objeto do banco de dados.

Alguns sistemas de banco de dados usam o comando ALTER, que permite ao usuário alterar um objeto, por exemplo, adicionando uma coluna a uma tabela existente.

Outros comandos *DDL*:

- CREATE TABLE
- CREATE INDEX
- CREATE VIEW
- ALTER TABLE
- ALTER INDEX
- DROP INDEX
- DROP VIEW

DML - Linguagem de Manipulação de Dados

O grupo [DML](#) (Data Manipulation Language - Linguagem de manipulação de dados). é utilizado para realizar inclusões, consultas, alterações e exclusões de dados presentes em registros. Estas tarefas podem ser executadas em vários registros de diversas tabelas ao mesmo tempo. Os comandos que realizam respectivamente as funções acima referidas são Insert, Update e Delete.

função	comandos SQL	descrição do comando	exemplo
inclusões	INSERT	é usada para inserir um registro (formalmente uma tupla) a uma tabela existente.	<code>INSERT INTO Pessoa (id, nome, sexo) VALUES;</code>
alterações	UPDATE	para mudar os valores de dados em uma ou mais linhas da tabela existente.	<code>UPDATE Pessoa SET data_nascimento = '11/09/1985' WHERE id_pessoa = 7</code>
exclusões	DELETE	permite remover linhas existentes de uma tabela.	<code>DELETE FROM pessoa WHERE id_pessoa = 7</code>

É possível inserir dados na tabela Area usando o `INSERT INTO`:

```
INSERT INTO Area (arecod, aredes) VALUES (100, "Informática"), (200,  
"Turismo"), (300, "Higiene e Beleza");
```

DQL - Linguagem de Consulta de Dados

Embora tenha apenas um comando, a DQL é a parte da SQL mais utilizada. O comando [SELECT](#) permite ao usuário especificar uma consulta ("query") como uma descrição do resultado desejado. Esse comando é composto de várias cláusulas e opções, possibilitando elaborar consultas das mais simples às mais elaboradas.

Função	Comandos SQL	Descrição do comando	Exemplo
consultas	<u>SELECT</u>	O Select é o principal comando usado em SQL para realizar consultas a dados pertencentes a uma tabela.	<code>Select * From Pessoa;</code>

DCL - Linguagem de Controle de Dados

Outro grupo é o [DCL](#) (Data Control Language - Linguagem de Controle de Dados). DCL controla 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.

Duas palavras-chaves da DCL:

- GRANT - autoriza ao usuário executar ou ajustar operações.
- REVOKE - remove ou restringe a capacidade de um usuário de executar operações.

DTL - Linguagem de Transação de Dados

Transação é um processo de acesso e tratamento de dados formado por vários comandos SQL e que não podem ser tratados de forma individual, atômica mas sim de forma grupal. Se um comando individual não consegue ser executado ou resulta em erro, todo o grupo de comandos deve ser cancelado e seus resultados, desfeitos.

Os comandos desse grupo são:

- BEGIN WORK (ou **BEGIN TRANSACTION**, dependendo do dialeto SQL) pode ser usado para marcar o começo de uma transação de banco de dados que pode ser completada ou não.
- COMMIT finaliza uma transação dentro de um sistema de gerenciamento de banco de dados.
- ROLLBACK faz com que as mudanças nos dados existentes desde o último COMMIT ou ROLLBACK sejam descartadas.

COMMIT e ROLLBACK interagem com áreas de controle como transação e locação. Ambos terminam qualquer transação aberta e liberam qualquer cadeado ligado a dados. Na ausência de um BEGIN WORK ou uma declaração semelhante, a semântica de SQL é dependente da implementação.

Linguagem de Programação

Além dos grupos acima, ainda existem comandos de programação “estruturada”, como **if**, **while**, **declare**, **fetch**, que permitem desenvolver procedimentos (métodos) que ficam armazenados no banco de dados e podem ser invocados (chamados) por aplicações do lado do cliente (usuário final) do sistema de banco de dados. A aplicação chama esse procedimento e ele é executado internamente no servidor de banco de dados. Essa abordagem evita que dados tenham que trafegar pela rede até o cliente, e que o processador da máquina cliente tenha que ser usado para processar esses dados. Ao final do processamento, os dados resultantes não trafegam pela rede desde o cliente para de volta ao servidor, pois sempre ficaram no servidor e lá foram processados.

Por outro lado, essa abordagem aumenta o gasto de memória e tempo de CPU do servidor de banco de dados, já que este tem que executar tais procedimentos.

Essas linguagens não são muito padronizadas, sendo que cada fornecedor tem dialetos bastante diferentes. Por exemplo, T-SQL do SQL Server, PL/SQL de Oracle.

Referências:

[https://pt.wikipedia.org/wiki/SQL#Subconjuntos do SQL](https://pt.wikipedia.org/wiki/SQL#Subconjuntos_do_SQL)

[Vídeo adicional - Introdução ao SQL](#)

4.2. DDL e Tipos de dados dos atributos

Os tipos básicos de dados disponíveis para atributos incluem os tipos numéricos, cadeias de caracteres, cadeiras de bits, Boolean, data e tempo.

Tipos de Dados String:

Tipo de Dado	Descrição	Tamanho máximo	Armazenamento
char(n)	Cadeia de caracteres ASCII de tamanho fixo	8.000 caracteres	Tamanho que foi definido
varchar(n)	Cadeia de caracteres ASCII de tamanho variável	8.000 caracteres	2 bytes + número de caracteres
varchar(max)	Cadeia de caracteres ASCII de tamanho variável	1.073.741.824 caracteres	2 bytes + número de caracteres
text	Cadeia de caracteres ASCII de tamanho variável	2GB de dados texto	4 bytes + número de caracteres
nchar	Cadeia de caracteres Unicode de tamanho fixo	4.000 caracteres	Tamanho que foi definido x 2
nvarchar	Cadeia de caracteres Unicode de tamanho variável	4.000 caracteres	
nvarchar(max)	Cadeia de caracteres Unicode de tamanho variável	536.870.912 caracteres	
ntext	Cadeia de caracteres Unicode de tamanho variável	2GB de dados texto	
binary(n)	Cadeia de binários de tamanho fixo	8.000 bytes	
varbinary	Cadeia de binários de tamanho variável	8.000 bytes	
varbinary(max)	Cadeia de binários de tamanho variável	2GB	
image	Cadeia de binários de tamanho variável	2GB	

Tipos de Dados Numéricos:

Tipo de Dados	Descrição	Armazenamento
bit	Inteiro que pode ser 0, 1, ou NULL	
tinyint	Permite números naturais de 0 a 255	1 byte
smallint	Permite números inteiros entre -32,768 e 32,767	2 bytes
int	Permite números inteiros entre -2,147,483,648 e 2,147,483,647	4 bytes
bigint	Permite números inteiros entre -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807	8 bytes
decimal(p,s)	Números com Precisão e Escala fixas: Permite números entre $-10^{38}+1$ até $10^{38}-1$. O parâmetro p indica o número total máximo de dígitos que podem ser armazenados (tanto à esquerda quanto à direita do ponto decimal). p deve ser um valor de 1 a 38, sendo que o default é 18. O parâmetro s indica o número máximo de dígitos armazenados à direita do ponto decimal; s deve ser um valor de 0 a p, sendo que o valor default é 0	5-17 bytes
numeric(p,s)	O mesmo que decimal(p,s)	5-17 bytes
smallmoney	Dados monetários de -214.748,3648 até 214.748,3647	4 bytes
money	Dados monetários de -922.337.203.685.477,5808 até 922.337.203.685.477,5807	8 bytes
float(n)	Dados numéricos de ponto flutuante de -1.79^{308} até 1.79^{308} . O parâmetro n indica se o campo deve armazenar 4 ou 8 bytes. float(24) armazena um campo de 4 bytes e float(53) armazena um campo de 8 bytes. O valor padrão de n é 53.	4 ou 8 bytes
real	Dados numéricos de ponto flutuante de -3.4^{38} até 3.4^{38} .	4 bytes

Dados de tipo Data e Tempo:

Tipo de Dados	Descrição	Armazenamento
datetime	Desde 1 de Janeiro de 1753 até 31 de dezembro de 9999 com uma precisão de 3.33 milissegundos	8 bytes
datetime2	Desde 1 de Janeiro de 0001 até 31 de dezembro de 9999 com precisão de 100 nanossegundos	6-8 bytes
smalldatetime	De 1 de Janeiro de 1900 até 6 de Junho de 2079 com precisão de 1 minuto	4 bytes
date	Armazena somente uma data. From January 1, 0001 to December 31, 9999	3 bytes
time	Armazena um horário somente até uma precisão de 100 nanossegundos	3-5 bytes
datetimeoffset	O mesmo que datetime2 com a adição de um deslocamento de zona de fuso horário	8-10 bytes
timestamp	Armazena um único número que é automaticamente atualizado pelo servidor de banco de dados cada vez que uma linha de dados é criada ou alterada. O valor de um timestamp é baseado em um relógio interno e não corresponde a um horário real. Cada tabela somente pode ter um único atributo de tipo timestamp.	

Outros tipos de dados:

Tipo de Dados	Descrição
sql_variant	Armazena até 8.000 bytes de dados de vários tipos, exceto texto, ntext e timestamp
uniqueidentifier	Armazena um Identificador Globalmente Único (globally unique identifier - GUID)
xml	Armazena dados formatados em XML, com tamanho máximo de 2GB
cursor	Armazena uma referência a um cursor (tabela temporária) usada para operações do banco de dados
table	Armazena um result-set para processamento posterior.

Referências:

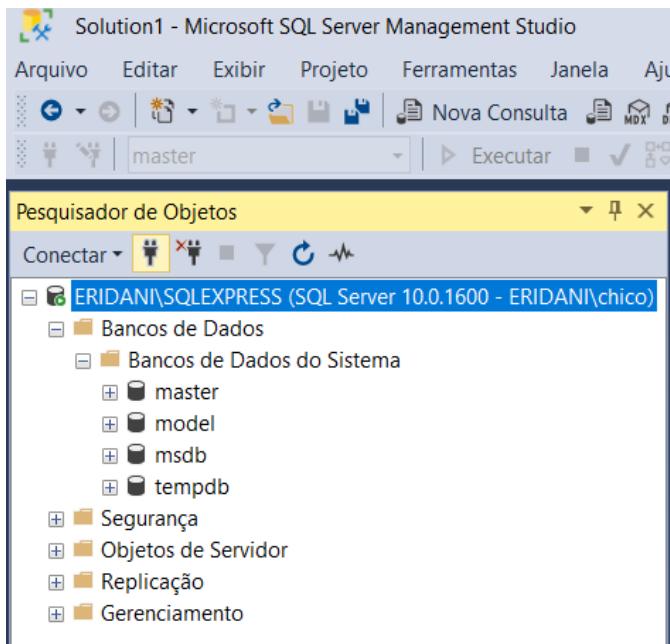
https://www.w3schools.com/sql/sql_datatypes.asp

[Vídeo adicional – SGBR e Tipos de Dados](#)

4.3. Criando um Banco de Dados, suas tabelas e relacionamentos no Servidor

Criação de um Banco de Dados no Sql Server

Depois de conectar-se ao servidor de banco de dados usando o Sql Server Management Studio (doravante chamado de SSMS), você pode começar a criar seus bancos de dados e configurá-los. Observe a figura abaixo:



Nessa figura vemos o Pesquisador de Objetos do SSMS. Ele nos mostra todos os objetos, de todos os bancos de dados já criados nesse servidor. Neste caso, ainda não temos nenhum banco de dados a não ser aqueles que são criados e configurados automaticamente durante a instalação do Sql Server (nessa figura, o Sql Server Express).

Esses bancos de dados são usados para controlar o acesso de usuários aos demais bancos de dados e fornecer todas as demais funcionalidades que um servidor de banco de dados oferece. Assim, esses bancos não devem ser, normalmente, manipulados por nós, desenvolvedores.

Há, também, objetos de segurança, como as contas de usuários criadas no servidor do banco de dados (diferentes das contas de usuários da rede dos usuários) e outros.

Para criar um banco de dados do sistema que você estiver desenvolvendo, abra uma janela de consulta, pressionando o botão [Nova consulta]. Aparecerá a janela de digitação de texto que vemos ao lado:

Nessa janela, digite o comando abaixo:

```
create database Empresa
```

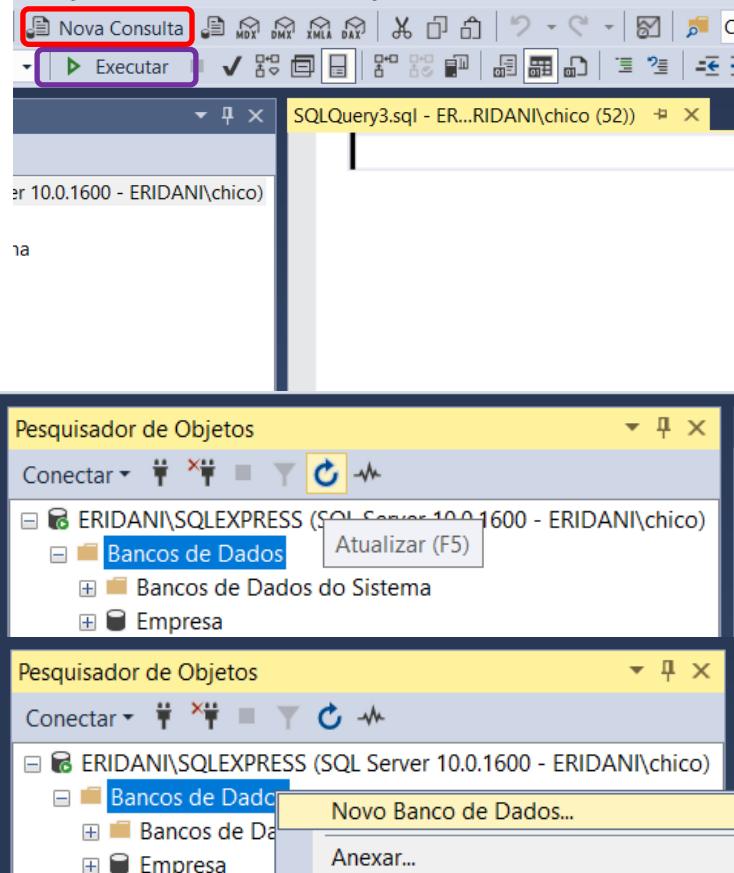
e pressione o botão [Executar] ou a tecla [F5].

Após isso, o banco de dados chamado Empresa será criado, mas ele ainda não apareceu no Pesquisador de Objetos.

Para isso, você deverá clicar no item Banco de Dados do Pesquisador e pressionar o botão [Atualizar] da janela.

Outra maneira de criar um banco de dados é clicar com o botão direito no item Bancos de Dados e selecionar a opção [Novo Banco de Dados...] do menu suspenso que aparecerá.

Digite o nome do banco de dados na janela que será exibida e pressione o botão [Ok].



[Vídeo adicional - criação de um banco de dados no SSMS](#)

Banco de Dados Empresa

Em nosso estudo de bancos de dados físicos e de um servidor de bancos de dados, usaremos um banco de dados de uma pequena empresa, cujo dicionário de dados vemos abaixo:

Empregado

Atributo	Tipo	Tam.	Restrições	Descrição	Referencia
prenome	varchar	15	not null	Prenome do Empregado	
inicialMeio	char	1	Not null	Letra do nome intermediário	
sobrenome	varchar	15	Not null	Sobrenome do empregado	
numSegSocial	char	9	PK, not null	Número de seguridade social, identifica o empregado	
dataNascimento	Date		Null	Nascimento do empregado	
endereco	varchar	30	Null	Endereço do empregado	
sexo	char	1	Null	Sexo do empregado	
salario	decimal	10,2	Null	Salário bruto do empregado	
super_numSegSocial	char	9	FK, not null	Número de seguridade social, identifica o chefe deste empregado	Empregado
numDept	int		FK, not Null	Número do departamento onde este empregado está lotado	Departamento

Departamento

Atributo	Tipo	Tam.	Restrições	Descrição	Referencia
nomeDept	varchar	15	not null, único	Nome do Departamento	
numDept	int		PK, not Null	Identificação do departamento	
gerente_numSegSocial	char	9	FK, Not null	Número de seguridade social do gerente deste departamento	Empregado
gerente_dataIncial	Date			Data de início do gerente	

Dept_Locais

Atributo	Tipo	Tam.	Restrições	Descrição	Referencia
numDept	int	32	FK, not Null	Identificação do departamento	Departamento
localDept	varchar	15	not Null	Nome da cidade do departamento	

chave primária composta

Projeto

Atributo	Tipo	Tam.	Restrições	Descrição	Referencia
nomeProjeto	Varchar	15	not null, único	Nome do projeto	
numProjeto	int		PK, not Null	Identificação do projeto	
localProjeto	varchar	15	null	Local onde o projeto é realizado	
numDept	int		FK, not null	Departamento responsável	Departamento

Trabalha_em

Atributo	Tipo	Tam.	Restrições	Descrição	Referencia
----------	------	------	------------	-----------	------------

<u>numSeqSocial</u>	char	9	FK, not null	Número de seguridade social de empregado alocado no projeto	Empregado
<u>numProjeto</u>	int		FK, not Null	Número que identifica o projeto	Projeto
Horas	decimal	3,1	Not null	Número de horas do empregado aplicadas no projeto	

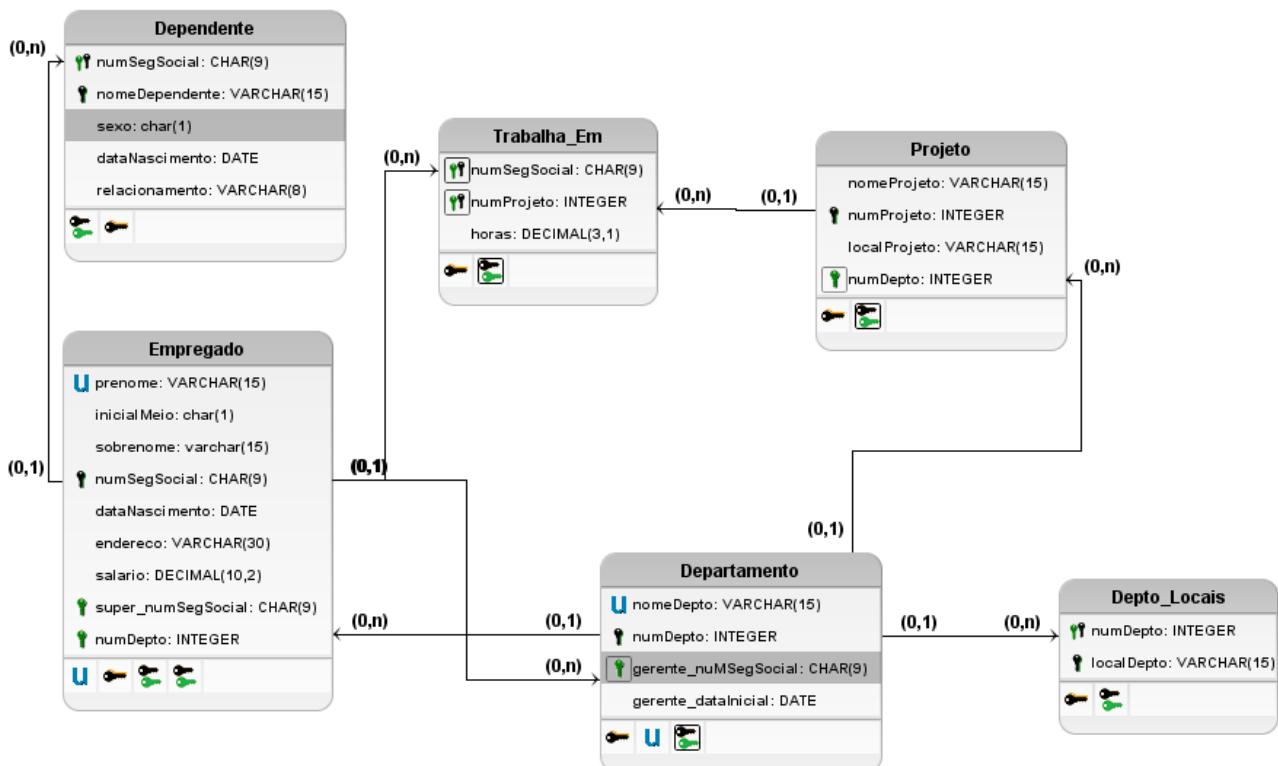
Chave primária composta

Dependente

Atributo	Tipo	Tam.	Restrições	Descrição	Referencia
<u>numSeqSocial</u>	char	9	FK, not null	Número de seguridade social de empregado responsável pelo dependente	Empregado
<u>nomeDependente</u>	varchar	15	Not null	Nome do dependente	
<u>sexo</u>	char	1	null	Sexo do dependente	
<u>dataNascimento</u>	Date		null	Data de Nascimento do dependente	
<u>relacionamento</u>	varchar	8	Null	Tipo de relacionamento	

Chave primária composta

O diagrama lógico, feito com o brModelo, vem abaixo:



Estude as tabelas, seus atributos, chaves primárias e estrangeiras, que definem os relacionamentos entre as tabelas. Será importante para as atividades de criação desse banco de dados e inclusão das informações nele.

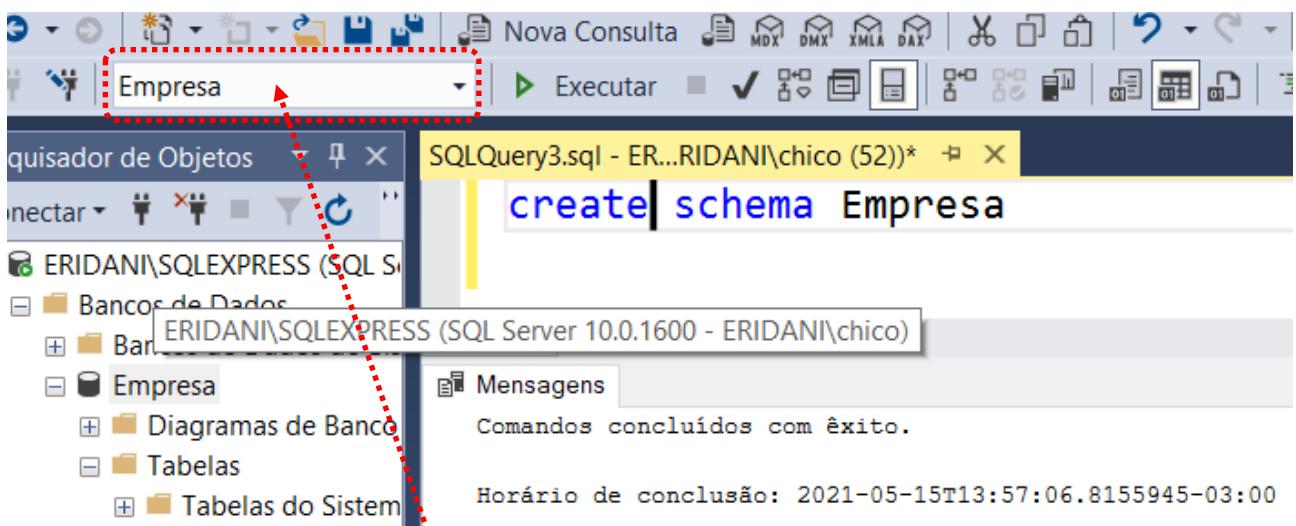
ESQUEMA de BANCO de DADOS

O esquema de um banco de dados é o agrupamento de objetos que, associados, representam no servidor **físico** a modelagem de um banco de dados específico. Esses objetos são, por exemplo, as tabelas, os campos, as chaves primárias e estrangeiras, índices, triggers, visualizações, funções e procedimentos armazenados, dentre outros.

O esquema padrão do Sql Server é o dbo, que significa **DataBase Owner**. Todo objeto de banco de dados criado sem que um nome de esquema seja atribuído é colocado no esquema dbo. Assim, esse objeto acaba ficando misturado aos demais, sem distinção a qual sistema ele faz parte. Isso pode ser complicado, pois num mesmo servidor podemos ter um único banco de dados atendendo a vários sistemas computacionais distintos e que, muitas vezes, não tem qualquer relação ou contato entre si. Por exemplo, num mesmo banco de dados físico podemos ter tabelas para contabilidade, para vendas, compras, recursos humanos, etc, e podemos usar esquemas para agrupar as tabelas de cada sistema, mantendo-as dentro do mesmo banco de dados físico mas tendo o esquema como elemento que as agrupa funcionalmente.

Criar um esquema no banco de dados específico para cada sistema computacional é uma boa estratégia para visualizar apenas os objetos que façam parte desse sistema específico.

Para criar um esquema no Sql Server, devemos primeiramente conectar ao servidor de banco de dados e, depois disso, abrir uma janela de consulta, clicando no botão [Nova consulta], selecionar o banco de dados ao qual esse esquema será associado (**Empresa** ou **Emp**, por exemplo), digitar e executar o comando **Create Schema**, como vemos na figura abaixo:



Não se esqueça de selecionar o banco de dados, senão o esquema será criado no banco de dados do Sistema Master, ao qual não temos acesso.

Quando formos criar os objetos de um banco de dados, podemos associar cada um deles a um esquema específico, criado com o comando acima. Por exemplo, ao criarmos uma tabela no banco de dados, usamos o comando **Create Table nomeDoEsquema.Tabela**.

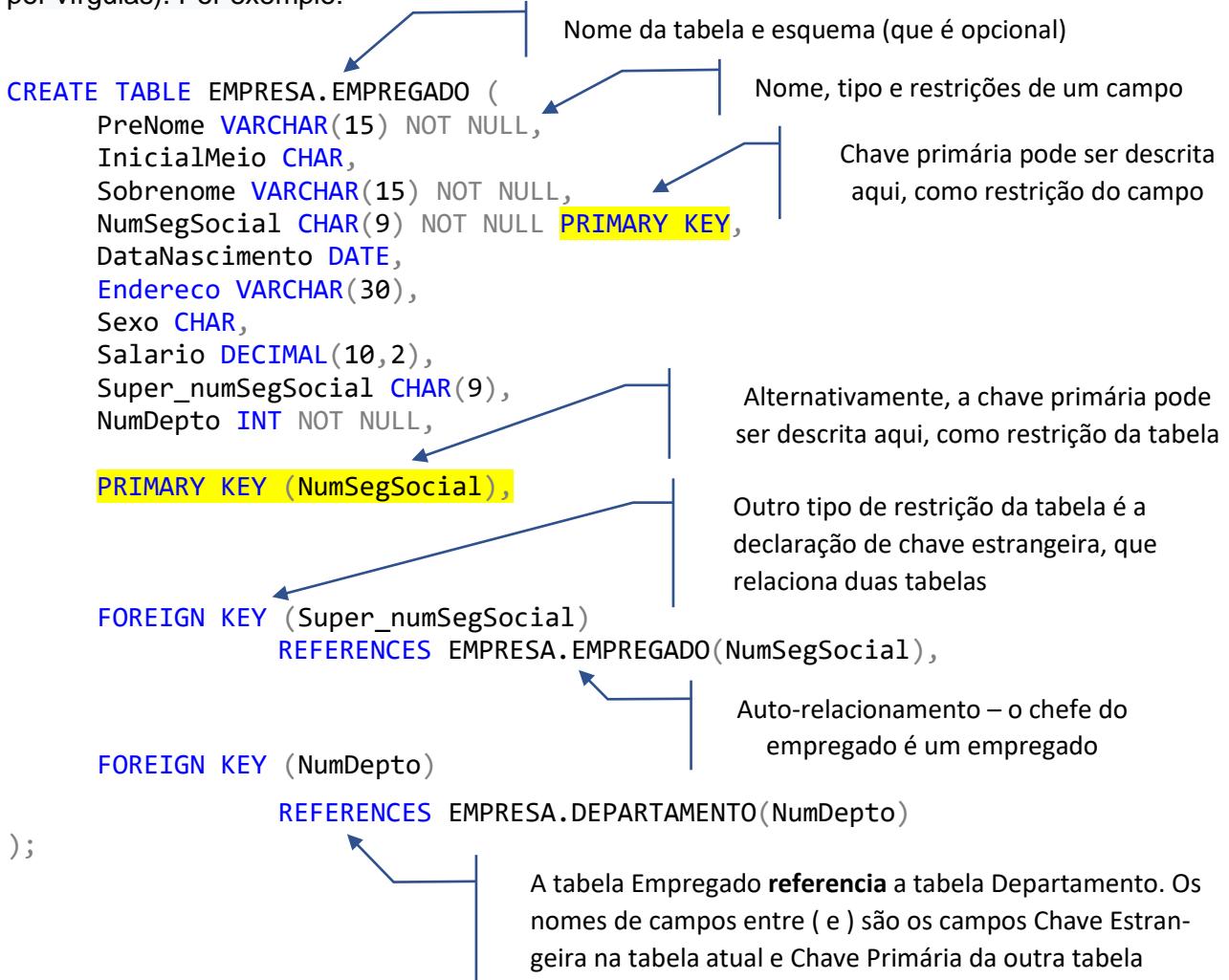
Criação de Tabelas

O comando **CREATE TABLE** é usado para especificar uma nova tabela, dando-lhe um nome e especificando seus atributos e restrições iniciais. Os atributos são especificados primeiro, e cada atributo recebe um nome, um tipo de dados para especificar seu domínio de valores e quaisquer restrições de atributo, como NOT NULL. A chave primária, as chaves estrangeiras que referenciam outras tabelas e restrições de integridade podem ser especificadas na instrução **CREATE TABLE** após os atributos terem sido declarados ou podem ser adicionados posteriormente usando o comando **ALTER TABLE**.

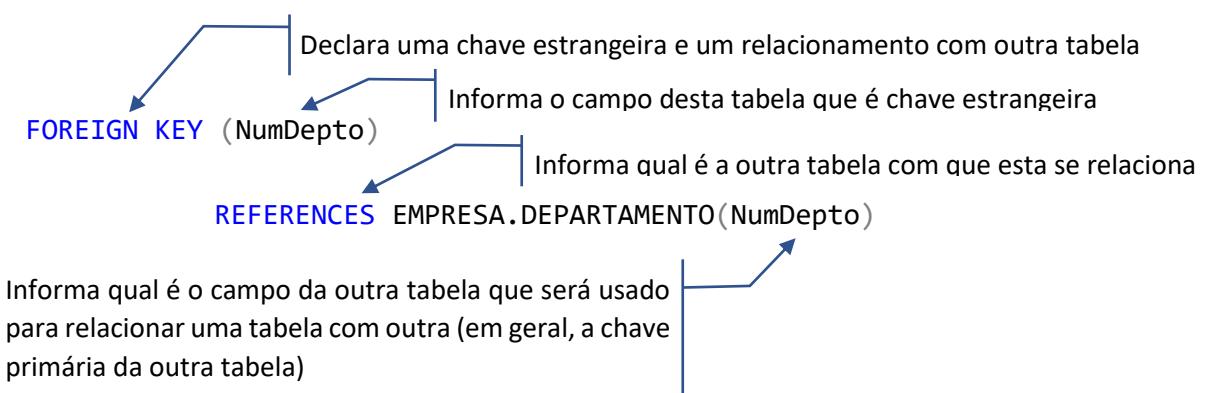
Esse comando tem a seguinte forma geral:

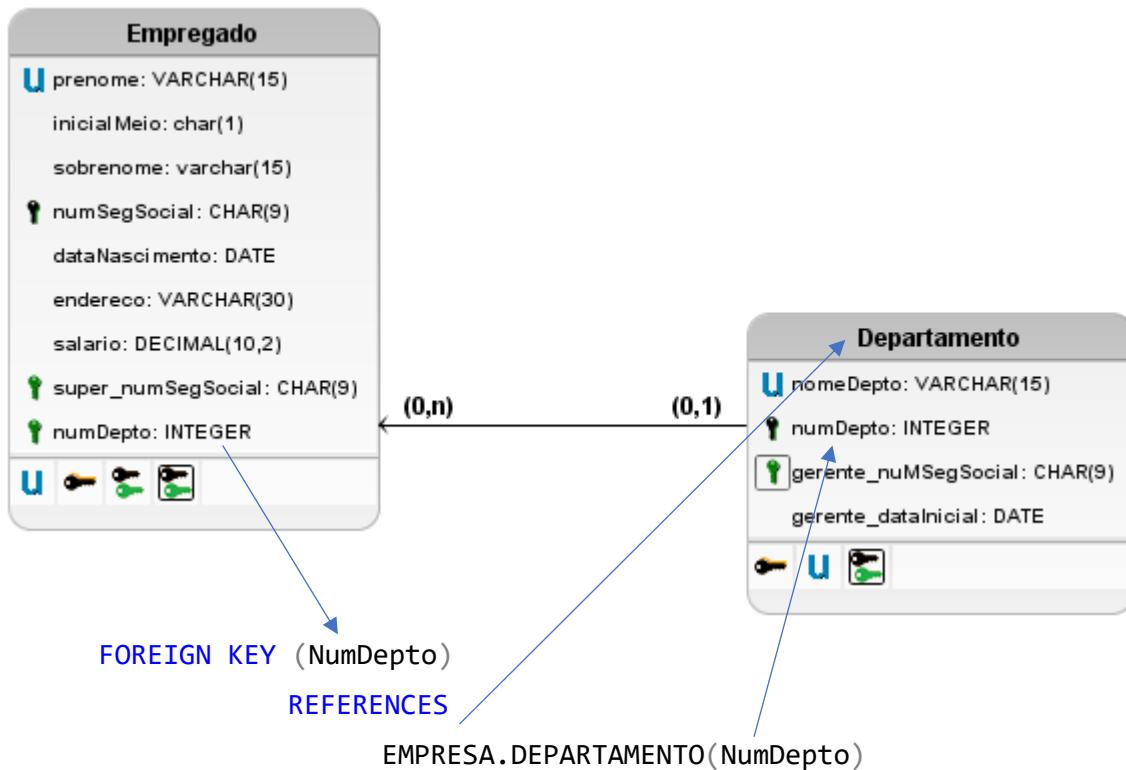
```
CREATE TABLE <nome da tabela>
(
    <nome de campo> <tipo do campo> [ <restrições do atributo> ]
    , <nome de campo> <tipo do campo> [ <restrições do atributo> ] 
    [ <Restrições da tabela> { , <Restrições da tabela> } ]
)
```

Nessa descrição, o que está entre [e] é uma cláusula opcional no comando. O que estiver entre { e } é repetitivo, o seja, pode ocorrer 1 ou mais vezes (note que esses itens repetitivos são separados por vírgulas). Por exemplo:



Portanto, a cláusula Foreign Key é a maneira como a linguagem SQL indica para o servidor de banco de dados qual campo da tabela que estamos descrevendo é uma chave estrangeira. Essa chave estrangeira remete (em geral) a uma chave primária de outra tabela. Assim, a tabela atual referencia outra tabela. O campo da outra tabela usado como campo de junção de informações é colocado entre parênteses após o nome da tabela referenciada:





É importante notar que, se você optar por definir as chaves estrangeiras dentro do comando Create Table, algumas definições poderão causar erros por serem especificados através de **referências circulares** ou porque elas se referem para uma **tabela que ainda não foi criada**.

Por exemplo, a chave estrangeira NumDept na tabela EMPREGADO refere-se à tabela DEPARTAMENTO, que ainda não foi criada:

Observe a figura abaixo, que mostra uma mensagem do servidor de banco de dados Sql Server, acessado pelo SSMS com o comando Create Table acima e que produziu o erro citado anteriormente:

```

Super_numSegSocial CHAR(9),
NumDept INT NOT NULL,
PRIMARY KEY (NumSegSocial),
FOREIGN KEY (Super_numSegSocial) REFERENCES EMPREGADO(NumSegSocial),
FOREIGN KEY (NumDept) REFERENCES DEPARTAMENTO(NumDept)
);

```

100 % Mensagens

Mensagem 1767, Nível 16, Estado 0, Linha 2
Foreign key 'FK_EMPREGADO_Super_07020F21' references invalid table 'EMPREGADO'.
Mensagem 1750, Nível 16, Estado 0, Linha 2
Could not create constraint. See previous errors.

Horário de conclusão: 2021-05-15T14:03:22.7205107-03:00

Em algumas versões do Sql Server, referências circulares feitas no momento da criação da tabela poderão acusar erro, também. Por exemplo, a chave estrangeira **Super_numSegSocial** na tabela EMPREGADO é uma **referência circular** porque se refere a essa mesma tabela que, durante o Create Table, ainda não está criada. Em algumas versões isso poderá ser impedido.

Para lidar com esse tipo de problema, essas **constraints** (restrições) podem ser deixadas fora da instrução CREATE TABLE inicial e, posteriormente, adicionadas usando o ALTER TABLE (que estudaremos logo mais).

Para criarmos as tabelas do nosso banco de dados físico de estudo (Empresa), usaremos as definições do Diagrama Lógico ER e do dicionário de dados que apresentamos anteriormente.

Após digitar o comando abaixo na janela de edição, pressione o botão [Executar]:

```
CREATE TABLE Empresa.EMPREGADO
(
    PreNome VARCHAR(15) NOT NULL,
    InicialMeio CHAR,
    Sobrenome VARCHAR(15) NOT NULL,
    NumSegSocial CHAR(9) NOT NULL,
    DataNascimento DATE,
    Endereco VARCHAR(30),
    Sexo CHAR,
    Salario DECIMAL(10,2),
    Super_numSegSocial CHAR(9),
    NumDept INT NOT NULL,
    PRIMARY KEY (NumSegSocial)
);
```

E, depois de clicarmos no ícone do banco de dados Empresa no Pesquisador de Objetos e clicar no botão [Atualizar], teríamos o resultado abaixo, onde vemos que a mensagem resultante da execução informa que houve êxito e podemos ver a tabela criada ao lado esquerdo (EMPREGADO), na lista de tabelas:

The screenshot shows the SQL Server Management Studio interface. On the left, the 'Pesquisador de Objetos' (Object Explorer) displays the database structure. Under the 'Bancos de Dados' node, the 'Empresa' database is selected. Inside 'Empresa', the 'Tabelas' node is expanded, and the 'Tabelas do Sistema' node is selected. Within this node, the 'Empresa.EMPREGADO' table is highlighted with a red rectangle. On the right, the 'SQLQuery3.sql - ER...RIDANI\chico (52)*' query editor window contains the SQL code for creating the table. The code is as follows:

```
CREATE TABLE Empresa.EMPREGADO
(
    PreNome VARCHAR(15) NOT NULL,
    InicialMeio CHAR,
    Sobrenome VARCHAR(15) NOT NULL,
    NumSegSocial CHAR(9) NOT NULL,
    DataNascimento DATE,
    Endereco VARCHAR(30),
    Sexo CHAR,
    Salario DECIMAL(10,2),
    Super_numSegSocial CHAR(9),
    NumDept INT NOT NULL,
    PRIMARY KEY (NumSegSocial)
);
```

Below the code, a message box is displayed with the text 'Comandos concluidos com êxito.' (Commands completed successfully.) This message is also highlighted with a red rectangle. At the bottom of the window, the status bar shows the completion time: 'Horário de conclusão: 2021-05-15T14:07:10.4670585-03:00'.

Outra possibilidade teria sido criar primeiramente a tabela DEPARTAMENTO, para que a referência que EMPREGADO faz a essa tabela na definição da chave estrangeira não acusasse erro. No entanto, se observarmos a definição da tabela DEPARTAMENTO, notamos que ela faz uma referência a EMPREGADO (indica o empregado que é o chefe do departamento). Assim, nesse caso específico, teríamos ainda de continuar declarando sem as duas referências em Empregado e sem a referência em Departamento, e corrigir isso depois de criadas as tabelas, usando o comando de SQL ALTER TABLE.

Quando você tiver vários comandos na janela de consulta, e quiser executar apenas um deles (ou mesmo um grupo deles, mas não todos), deverá selecionar com o mouse as linhas dos comandos que deseja executar e, depois disso, pressionar o botão {Executar} ou a tecla [F5].

Vamos agora criar a tabela **Departamento**, especificando a Foreign Key para Empregado, pois a tabela Empregado já existe. Digite o comando abaixo na janela de edição:

```
CREATE TABLE Empresa.DEPARTAMENTO
(
    NomeDept VARCHAR(15) NOT NULL,
    NumDept INT NOT NULL,
    Gerente_numSegSocial CHAR(9) NOT NULL,
    Gerente_dataInicial DATE,
    PRIMARY KEY (NumDept),
    UNIQUE (NomeDept),
    FOREIGN KEY (Gerente_numSegSocial)
        REFERENCES Empresa.EMPREGADO(NumSegSocial)
);

```

Primary Key Informa o nome do campo que é a chave primária desta tabela

Unique informa que o valor desse campo não pode ter repetições entre os vários registros da tabela

Foreign Key informa que o valor desse campo não pode ter repetições entre os vários registros da tabela

E aproveitamos para alterar a tabela Empregado e fazê-la referenciar Departamento:

```
ALTER TABLE Empresa.EMPREGADO
ADD FOREIGN KEY (Super_numSegSocial)
    REFERENCES Empresa.EMPREGADO(NumSegSocial),
    FOREIGN KEY (NumDept)
        REFERENCES Empresa.DEPARTAMENTO(NumDept);
```

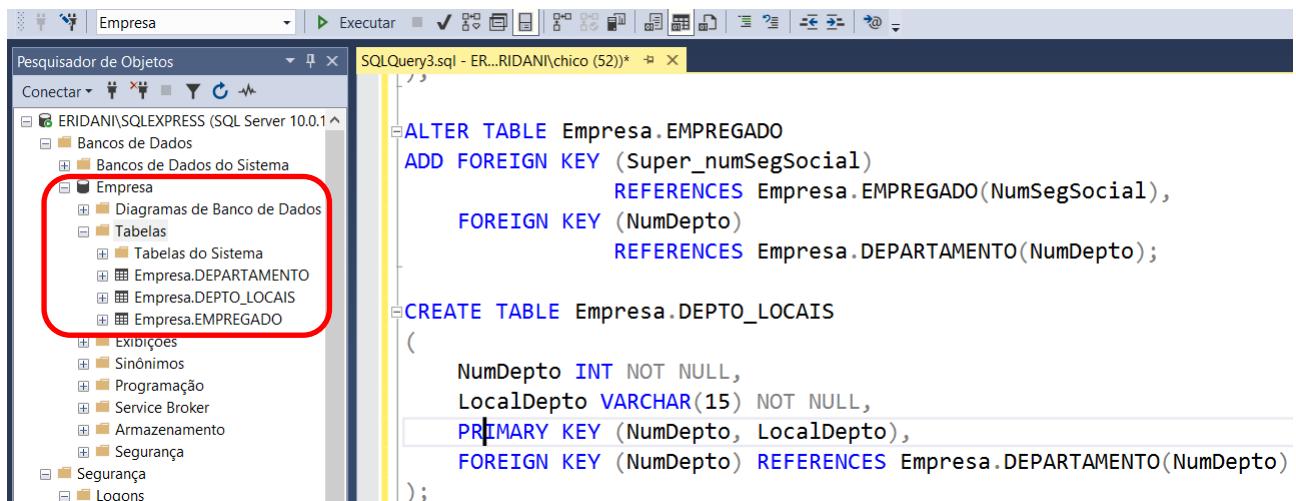
O comando **Alter Table** tem uma cláusula **ADD** que permite adicionar elementos a uma tabela. Podemos adicionar colunas (atributos), chaves estrangeiras, a chave primária (se ainda não o tivermos feito no CREATE TABLE).

Após digitar as linhas dos comandos Create Table e Alter Table na janela de edição, marque-os com o mouse ou teclado (selecione-os) e pressione o botão [Executar].

Vamos criar mais uma tabela, que relaciona Departamento com Locais onde se encontram. Após digitar o comando abaixo na janela de edição, **marque-o** e pressione o botão [Executar]:

```
CREATE TABLE Empresa.DEPTO_LOCAIS
(
    NumDept INT NOT NULL,
    LocalDept VARCHAR(15) NOT NULL,
    PRIMARY KEY (NumDept, LocalDept),
    FOREIGN KEY (NumDept) REFERENCES Empresa.DEPARTAMENTO(NumDept)
);
```

Chave primária composta por dois campos



Na figura acima, vemos as três tabelas que criamos. Posteriormente aprenderemos uma ferramenta visual que nos permitirá visualizar, também, os relacionamentos (dados pelas chaves estrangeiras) entre elas.

Vamos agora criar a tabela de Projetos. Digite o código abaixo, marque-o e pressione [Executar]:

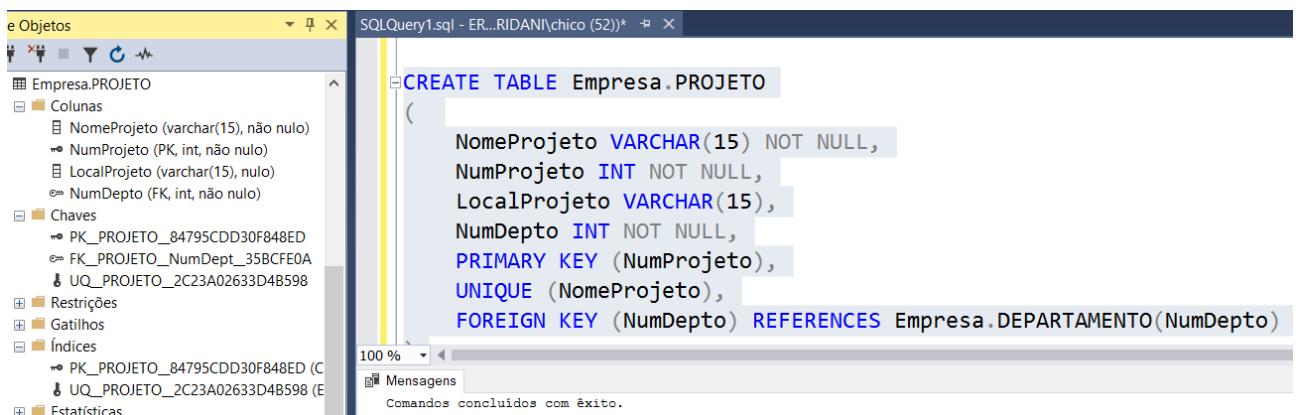
```

CREATE TABLE Empresa.PROJETO
(
    NomeProjeto VARCHAR(15) NOT NULL,
    NumProjeto INT NOT NULL,
    LocalProjeto VARCHAR(15),
    NumDept INT NOT NULL,
    PRIMARY KEY (NumProjeto),
    UNIQUE (NomeProjeto),
    FOREIGN KEY (NumDept) REFERENCES Empresa.DEPARTAMENTO(NumDept)
);

```

Se você não colocar o prefixo Empresa antes do nome das tabelas, elas serão criadas em outro esquema. Da mesma maneira, quando se declara uma Foreign Key, a tabela referenciada deve ter o prefixo de esquema, caso se tenha adotado seu uso na criação do banco de dados.

Após criar essa tabela, colocamos o cursor na coleção Tabelas do Pesquisador de Objetos e pressionamos a tecla [F5] para atualizar a visualização dessa janela. Procure a tabela PROJETO e expanda os itens Colunas, Chaves e Índices. Veremos o resultado como na figura abaixo:



Observe que temos a definição dos atributos (colunas) conforme foram descritos no comando CREATE TABLE. Também conseguimos ver a definição de chaves primária e estrangeira.

Temos agora a criação de mais uma tabela, com o código abaixo.

```
CREATE TABLE Empresa.TRABALHA_EM
(
    numSegSocial CHAR(9) NOT NULL,
    NumProjeto INT NOT NULL,
    Horas DECIMAL(3,1) NOT NULL,
    PRIMARY KEY (numSegSocial, NumProjeto),
    FOREIGN KEY (numSegSocial) REFERENCES Empresa.EMPREGADO(NumSegSocial),
    FOREIGN KEY (NumProjeto) REFERENCES Empresa.PROJETO(NumProjeto)
);
```

Chave primária composta por dois campos

Finalmente, temos a última tabela, chamada DEPENDENTE:

```
CREATE TABLE Empresa.DEPENDENTE
(
    NumSegSocial CHAR(9) NOT NULL,
    NomeDependente VARCHAR(15) NOT NULL,
    Sexo CHAR,
    DataNascimento DATE,
    Relacionamento VARCHAR(8),
    PRIMARY KEY (NumSegSocial , NomeDependente),
    FOREIGN KEY (NumSegSocial) REFERENCES Empresa.EMPREGADO(NumSegSocial)
);
```

Chave primária composta por dois campos

Agora já temos um banco de dados de exemplo criado. Note que essa é implementação física de um Banco de Dados, que podemos ver representada pela interface com o usuário providenciada pelo SSMS, ao lado:

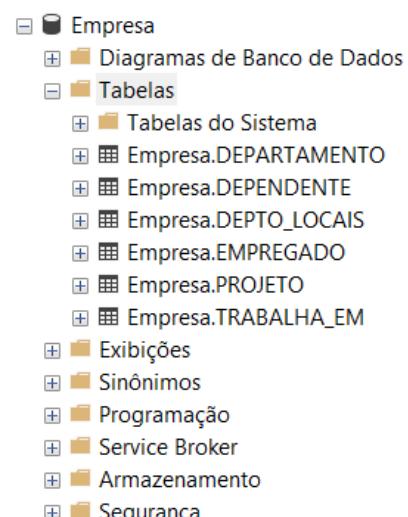
Posteriormente faremos inclusões de dados para podermos estudar os próximos comandos.

[Vídeo adicional - Constraints](#)

[Vídeo adicional – Criação de Tabelas](#)

[Vídeo adicional - Campos autoincremento](#)

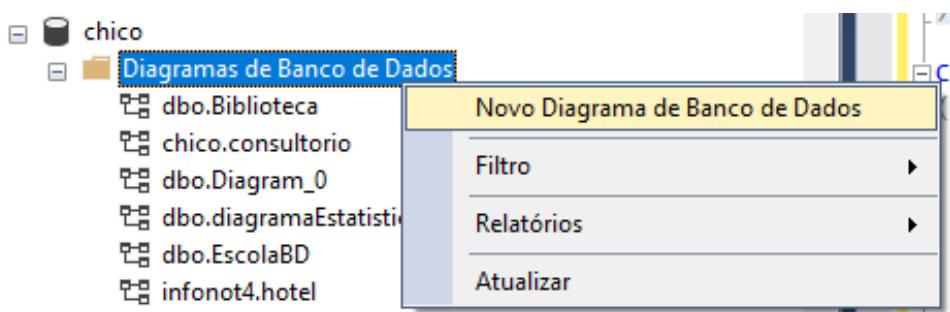
[Vídeo adicional - Alter e Drop Table](#)



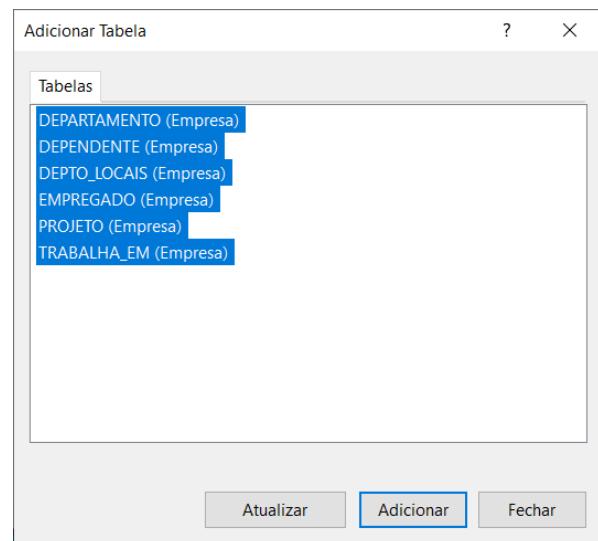
4.4. Diagrama do Banco de Dados

Antes de prosseguirmos, vamos criar uma visualização desse banco, de forma que possamos ver os relacionamentos entre as tabelas.

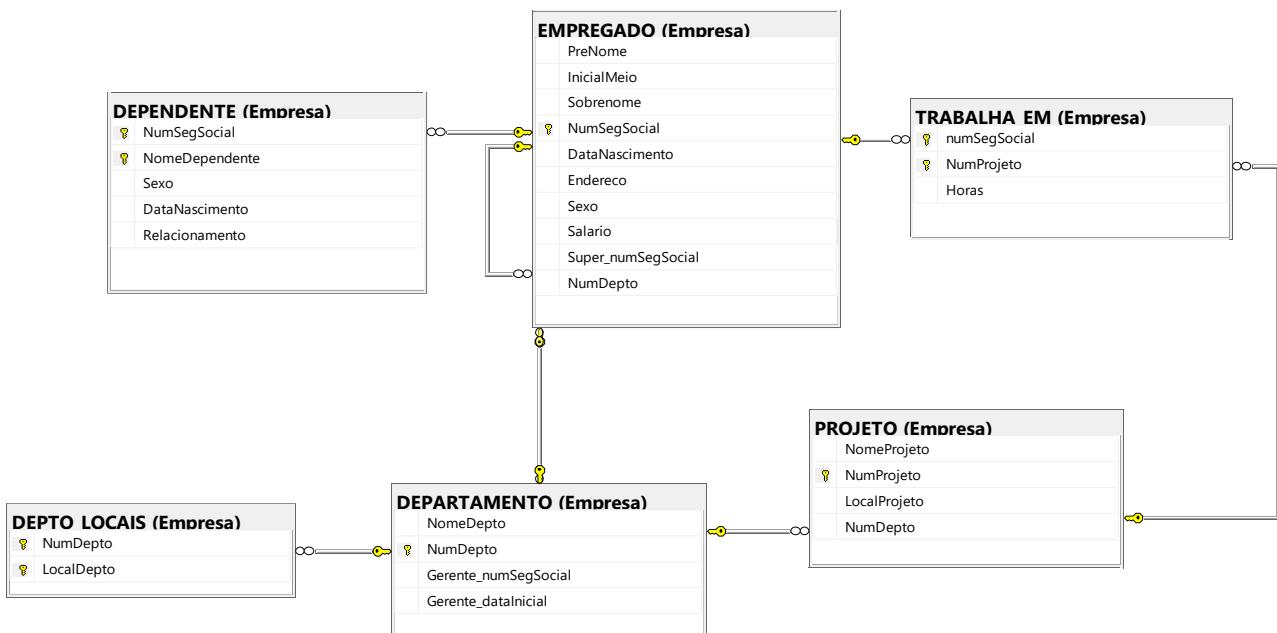
Para isso, no Pesquisador de Objetos, expanda o nome do seu banco de dados e clique, com o botão direito do mouse, na coleção Diagramas de Banco de Dados. Aparecerá um menu suspenso como o da figura abaixo. Selecione a opção Novo Diagrama de Banco de Dados.



Em seguida, devemos escolher as tabelas que fazem parte do nosso banco de dados e elas aparecerão no diagrama, como vemos ao lado.



Note que, na figura abaixo, as tabelas já foram reorganizadas uma a uma, bem como os relacionamentos, para aparecerem numa disposição mais adequada à compreensão dos relacionamentos.



As relações declaradas por meio das instruções CREATE TABLE são chamadas de tabelas base (ou relações de base); isso significa que a relação e suas tuplas são realmente criadas e armazenadas como um arquivo pelo DBMS.

As relações de base são diferenciadas das relações virtuais, criado por meio da instrução CREATE VIEW (que estudaremos no semestre que vem, em Bancos de Dados II), que pode ou não corresponder a um arquivo físico real.

Na próxima figura 4.7, abaixo, apresentamos com possíveis dados para as nossas tabelas:

EMPREGADO

PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNascimento	Endereco	Sexo	Salario	Super_numSegSocial	NumDept
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.00	888665555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.00	333445555	5
Igor	N	Camargo	494549454	1985-10-26	53 Dona Libânia,Campinas,SP	M	30000.00	888665555	6
Luis	F	Assis	512851285	1994-06-21	37 Reg Feijó, Campinas, SP	M	20000.00	888665555	6
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000.00	333445555	5
Cecilia	F	Kolonsky	677678989	1960-04-05	6357 Windy Lane, Katy, TX	F	28000.00	NULL	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.00	NULL	1
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.00	888665555	4
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.00	987654321	4
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000.00	987654321	4

DEPARTAMENTO

NomeDept	NumDept	Gerente_numSegSocial	Gerente_dataIncial
Sede	1	888665555	1981-06-19
Administração	4	987654321	1995-01-01
Pesquisa	5	333445555	1988-05-22
Tecn Informação	6	512851285	2020-04-29

DEPTO_LOCAIS

NumDept	LocalDept
1	Houston
4	Stafford
5	Belaire
5	Houston
5	Sugarland

TRABALHA_EM

numSegSocial	NumProjeto	Horas
123456789	1	32.5
123456789	2	7.5
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
453453453	1	20.0
453453453	2	20.0
494549454	20	25.0
512851285	10	20.0
666884444	3	40.0
888665555	20	40.0
987654321	20	15.0
987654321	30	20.0
987987987	10	35.0
987987987	30	5.0
999887777	10	10.0
999887777	30	30.0

PROJETO

NomeProjeto	NumProjeto	LocalProjeto	NumDept
ProdutoX	1	Bellaire	5
ProdutoY	2	Sugarland	5
ProdutoZ	3	Houston	5
Informatização	10	Stafford	4
Reorganização	20	Houston	1
Novos Benefíc	30	Stafford	4

DEPENDENTE

NumSegSocial	NomeDependente	Sexo	DataNascimento	Relacionamento
123456789	Alice	F	1988-12-30	Filha
123456789	Elizabeth	F	1967-05-05	Esposa
123456789	John	M	1988-01-04	Filho
333445555	Alice	F	1986-04-05	Filha
333445555	Franklin	M	1983-10-25	Filho
333445555	Joy	F	1958-05-03	Esposa
494549454	Marcos	M	2009-10-15	NULL
987654321	Abner	M	1942-02-28	Esposo

Figura 4.7

No SQL, os atributos em uma tabela base são considerados ordenados na sequência em que são especificados no comando CREATE TABLE. No entanto, as linhas (tuplas) não são consideradas ordenadas dentro de uma relação.

4.5. Comandos de Manipulação de Dados

4.5.1. Inserindo os dados das tabelas - Comando Insert

Vamos começar a inserir dados nas tabelas. Observando o diagrama do banco de dados, podemos notar que a tabela Departamento é a que menos depende de outras tabelas, então iniciaremos as inclusões por ela.

A forma geral do comando INSERT, usado para inserir registros nas tabelas, é a seguinte:

```
INSERT INTO <tabela> [<lista de campos>] VALUES (<lista de valores dos campos>)
```

O comando INSERT, na sua forma mais simples, é usado para adicionar um único registro (tupla) a uma tabela. Devemos especificar o nome da tabela e uma lista de valores para o registro. Os valores devem estar listados na mesma ordem em que os atributos correspondentes foram especificados no comando CREATE TABLE. Por exemplo, para adicionar uma nova tupla à tabela DEPARTAMENTO, podemos usar o comando abaixo:

```
Insert into Empresa.DEPARTAMENTO
values ('Pesquisa', 5, '333445555', Convert(date,'22/05/1988', 103))
```

Observe que não colocamos a <lista de campos>, porque ela é [opcional]. Quando usamos essa forma, sem a lista de campos, precisamos colocar valores para **todos os campos** que foram definidos para a tabela no momento de sua criação, e na ordem em que foram definidos.

Outra forma do comando Insert é com a **lista de campos**. No caso da tabela Departamento, teríamos o comando completo conforme o seguinte:

```
Insert into Empresa.DEPARTAMENTO
(nomeDepto, numDept, Gerente_numSegSocial, gerente_DataInicial)
values ('Pesquisa', 5, '333445555', Convert(date,'22/05/1988', 103))
```

Isso é útil se uma tabela tiver muitos atributos, mas apenas a alguns desses atributos forem atribuídos valores no novo registro. No entanto, os valores devem incluir todos os atributos com a especificação NOT NULL e nenhum valor padrão. Atributos com NULL permitido ou valores Default são os que podem ser deixados de fora. Por exemplo, para inserir um registro para um novo EMPREGADO, de quem conhecemos apenas os atributos PreNome, Sobrenome, NumDept e NumSegSocial, podemos usar

```
INSERT INTO Empresa.EMPREGADO (PreNome, Sobrenome, NumDept, NumSegSocial)
VALUES ('Richard', 'Marini', 4, '653298653')
```

Os atributos não especificados no comando acima são definidos como DEFAULT ou NULL, e os valores estão listados na mesma ordem em que os atributos estão listados no próprio comando INSERT.

Também é possível inserir em uma tabela vários registros separados por vírgulas em um único comando INSERT. Os valores dos atributos que formam cada registro devem estar entre parênteses. Por exemplo:

```
INSERT INTO Empresa.EMPREGADO (PreNome, Sobrenome, NumDept, NumSegSocial)
VALUES ('Richard', 'Marini', 4, '653298653'),
       ('Robert', 'Hatcher', 4, '980760540');
```

Um SGBD que implemente completamente o SQL deve oferecer suporte e impor toda a integridade restrições que podem ser especificadas no DDL. Por exemplo, se emitirmos o comando abaixo no banco de dados mostrado na [Figura 4.7](#), o DBMS deve rejeitar a operação porque não existe

nenhum registro de DEPARTAMENTO no banco de dados com NumDept = 2 e o campo NumDept de EMPREGADO, como chave estrangeira, faz uma referência a esse valor na tabela DEPARTAMENTO.

```
INSERT INTO Empresa.EMPREGADO (PreNome, Sobrenome, NumSegSocial, NumDept)
VALUES ('Robert', 'Hatcher', '980760540', 2);
```

A mensagem de erro do SQL Server seria:

```
Mensagem 547, Nível 16, Estado 0, Linha 1
A instrução INSERT conflitou com a restrição do FOREIGN KEY
"FK_EMPREGADO_NumDept_39E294A9". O conflito ocorreu no banco de dados
"chico", tabela "dbo.DEPARTAMENTO", column 'NumDept'.
```

A instrução foi finalizada

Da mesma forma, o comando abaixo seria rejeitado porque nenhum valor NumSegSocial é fornecido e esse campo é a chave primária, que não pode ser NULL.

```
INSERT INTO Empresa.EMPREGADO (PreNome, Sobrenome, NumDept)
VALUES ('Robert', 'Hatcher', 5);
```

```
Mensagem 515, Nível 16, Estado 2, Linha 12
Não é possível inserir o valor NULL na coluna 'NumSegSocial', tabela
'chico.dbo.EMPREGADO'; a coluna não permite nulos. Falha em INSERT.
A instrução foi finalizada.
```

Observamos nessa discussão que o servidor de banco de dados procura sempre manter a integridade referencial entre as tabelas, bem como as regras de integridade geral como, por exemplo, que a chave primária não pode ser nula. Se a chave primária fosse um campo numérico incremental, o erro acima não ocorreria, pois o próprio servidor se encarregaria de gerar o próximo valor do campo **Identity** e não o deixaria nulo.

Isso não quer dizer que as chaves incrementais sempre devam ser usadas. Seu uso depende das abordagens de projeto e da normalização.

Mas dissemos que íamos começar a inclusão de dados pela tabela Departamento, pois ela é a que menos depende das outras. No entanto, ela depende também, da própria tabela EMPREGADO e isso ocorre porque a tabela DEPARTAMENTO tem uma chave estrangeira que referencia Empregado (que indica o NUMSEGSOCIAL do supervisor desse departamento).

Vamos então tentar agora, na prática, incluir os registros de departamento. Digite o comando abaixo na janela de edição de consultas e o execute:

```
Insert into Empresa.DEPARTAMENTO
    (nomeDept, numDept, Gerente_numSegSocial, gerente_DataInicial)
values ('Pesquisa', 5,      '333445555', Convert(date, '22/05/1988', 103))
```

Aparecerá a mensagem de erro abaixo, na linguagem em que sua instalação foi feita:

```
Mensagem 547, Nível 16, Estado 0, Linha 1
A instrução INSERT conflitou com a restrição do FOREIGN KEY
"FK_DEPARTAMENTO_Mgr_s_37FA4C37". O conflito ocorreu no banco de dados "chico",
tabela "dbo.EMPREGADO", column 'NumSegSocial'.
A instrução foi finalizada.
```

Em inglês a mensagem será:

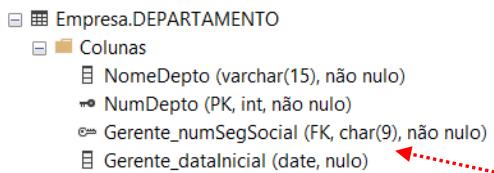
```
Mensagem 547, Nível 16, Estado 0, Linha 1
```

```
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_DEPARTAMENTO_Geren_1DE57479". The conflict occurred in database "Empresa",
table "Empresa.EMPREGADO", column 'NumSegSocial'.
```

The statement has been terminated.

Elá significa que ocorreu um **conflito de integridade referencial**. Esse conflito ocorreu com o campo NumSegSocial da tabela EMPREGADO, pois não existe, nessa tabela, um NumSegSocial com valor '**333445555**', já que ainda não conseguimos incluir nenhum empregado. Para incluir empregados, necessitamos que os departamentos onde eles trabalham já existam. Mas acabamos de ver que, neste momento, também não podemos incluir departamentos pois cada departamento precisa de um gerente que deveria existir na tabela EMPREGADO. Temos assim, um problema circular.

Para resolvê-lo, teremos de tornar o campo Gerente_numSegSocial de preenchimento não obrigatório. Se observarmos o comando de criação da tabela DEPARTAMENTO, notaremos que o campo Gerente_numSegSocial é obrigatório (NOT NULL):



```

CREATE TABLE DEPARTAMENTO
(
    NomeDept VARCHAR(15) NOT NULL,
    NumDept INT NOT NULL,
    Gerente_numSegSocial CHAR(9) NOT NULL,
    Gerente_dataInicial DATE,
    PRIMARY KEY (NumDept),
    UNIQUE (NomeDept),
    FOREIGN KEY (Gerente_numSegSocial)
        REFERENCES EMPREGADO(NumSegSocial)
);

```

Toda alteração da estrutura de uma tabela deve ser feita com o comando ALTER TABLE. Para alterar uma coluna no SQL Server, usamos a sintaxe abaixo:

```

ALTER TABLE nomeDaTabela
ALTER COLUMN nomeDaColuna tipoDeDados;

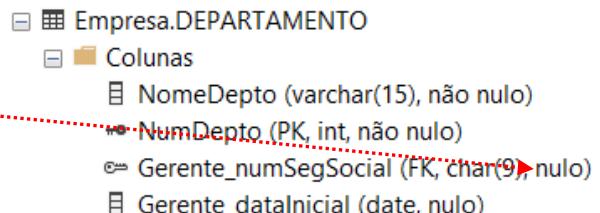
```

Assim, para alterar o campo Gerente_numSegSocial de modo a que aceite valores nulos, digitamos e executamos o comando abaixo:

```

ALTER TABLE Empresa.DEPARTAMENTO
ALTER COLUMN
    Gerente_numSegSocial char(9) Null;

```



Observe que o atributo Gerente_numSegSocial foi modificado de Not Null para Null. Agora, podemos fazer a inclusão de departamento e, depois que incluirmos os empregados, alterar o valor do campo Gerente_numSegSocial da tabela Departamento para que informem o gerente (cujo numSegSocial estará gravado na tabela Empregado) de cada departamento. Para isso, podemos usar um dos dois comandos abaixo:

```

Insert into Empresa.DEPARTAMENTO (nomeDept, numDept, gerente_DataInicial)
values ('Pesquisa', 5, Convert(date, '22/05/1988', 103))

```

ou

```
Insert into Empresa.DEPARTAMENTO
    (nomeDept, numDept, Gerente_numSegSocial, gerente_DataInicial)
values ('Pesquisa', 5, null, Convert(date,'22/05/1988', 103))
```

Agora, vamos incluir mais alguns registros, com o comando abaixo:

```
Insert
into Empresa.DEPARTAMENTO
    (nomeDept, numDept, gerente_DataInicial)
Values ('Administração', 4, Convert(date,'01/01/1995', 103)),
       ('Sede', 1, Convert(date,'19/06/1981', 103)),
       ('Tecnologia da Informação', 6, Convert(date,'29/04/2020', 103));
```

Esse comando acarretará um erro de cadeia de dados truncada, pois o nome do departamento (NomeDept) só poderá ter, no máximo, 15 caracteres e o nome do último departamento excede esse tamanho. A mensagem exibida vem abaixo, com a versão em inglês abaixo da em português:

Mensagem 8152, Nível 16, Estado 14, Linha 4
 Dados de cadeia ou binários seriam truncados.
 String or binary data would be truncated.
 A instrução foi finalizada.

Assim, teremos de abreviá-lo para caber em 15 caracteres. Essa é outra verificação de integridade que o servidor de banco de dados realiza. O comando com a abreviação do nome do departamento de TI vem abaixo. Digite-o e execute. Passaremos a ter 4 registros nessa tabela.

```
Insert
into Empresa.DEPARTAMENTO
    (nomeDept, numDept, gerente_DataInicial)
Values ('Administração', 4, Convert(date,'01/01/1995', 103)),
       ('Sede', 1, Convert(date,'19/06/1981', 103)),
       ('Tecn Informação', 6, Convert(date,'29/04/2020', 103));
```

Para consultar os registros da tabela Departamento, digite e execute o comando abaixo:

```
Select * from Empresa.DEPARTAMENTO
```

A execução desse comando produzirá um result set (conjunto de resultados) mostrado abaixo:

	NomeDept	NumDept	Gerente_numSegSocial	Gerente_dataInicial
1	Sede	1	NULL	1981-06-19
2	Administração	4	NULL	1995-01-01
3	Pesquisa	5	NULL	1988-05-22
4	Tecn Informação	6	NULL	2020-04-29

Podemos agora incluir os registros dos empregados e, depois, alterar os valores do campo Gerente_numSegSocial para cada registro de Departamento.

```
INSERT INTO Empresa.EMPREGADO
(PreNome, InicialMeio, Sobrenome, NumSegSocial, DataNascimento, Endereco,
Sexo, Salario, Super_numSegSocial, NumDept)
VALUES
('John', 'B', 'Smith', '123456789', '09/01/1965',
'731 Fondren, Houston, TX', 'M', 30000, '333445555', 5),
('Franklin', 'T', 'Wong', '333445555', '08/12/1955', '638 Voss, Houston, TX',
'M', 40000, '888665555', 5);
```

Ao executarmos esses comandos, teremos novamente erros de integridade referencial, pois quando o primeiro registro é inserido, o campo Super_numSegSocial contém um valor ('333445555') que ainda não foi inserido, mesmo sendo o registro seguinte na relação desse comando Insert. Isso ocorre porque a tabela Empregado possui um **auto-relacionamento**, onde o campo Super_numSegSocial referencia o numSegSocial do chefe desse empregado.

```
Mensagem 547, Nível 16, Estado 0, Linha 27
The INSERT statement conflicted with the FOREIGN KEY SAME TABLE constraint
"FK_EMPREGADO_Super_1ED998B2". The conflict occurred in database "Empresa",
table "Empresa.EMPREGADO", column 'NumSegSocial'.
The statement has been terminated.
```

Portanto, teremos de também alterar esse campo na tabela Empregado, para que ele aceite valores nulos. Depois de fazer isso, faremos as inclusões e posteriormente modificaremos os registros para que esse campo passe a indicar o supervisor de cada empregado. O comando abaixo modifica esse campo para que aceite valores nulos durante inclusão e alteração. Execute-o:

```
ALTER TABLE Empresa.EMPREGADO
ALTER COLUMN Super_numSegSocial char(9) Null;
```

Execute o comando acima e, depois, tente novamente executar os comandos de inserção da página anterior, **mas colocando o valor null** no lugar das strings de Super_numSegSocial:

```
INSERT INTO Empresa.EMPREGADO
(PreNome, InicialMeio, Sobrenome, NumSegSocial, DataNascimento, Endereco,
Sexo, Salario, Super_numSegSocial, NumDept)
VALUES
('John', 'B', 'Smith', '123456789', '09/01/1965', '731 Fondren, Houston, TX',
'M', 30000, null, 5),
('Franklin', 'T', 'Wong', '333445555', '08/12/1955', '638 Voss, Houston, TX',
'M', 40000, null, 5);
```

Teremos o resultado abaixo quando executarmos Select * from Empresa.EMPREGADO e Select * from Empresa.Departamento:

	PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNascimento	Endereco	Sexo	Salario	Super_numSegSocial	NumDept
1	John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000.00	NULL	5
2	Franklin	T	Wong	333445555	1955-08-12	638 Voss, Houston, TX	M	40000.00	NULL	5

	NomeDept	NumDept	Gerente_numSegSocial	Gerente_dataNcial
1	Sede	1	NULL	1981-06-19
2	Administração	4	NULL	1995-01-01
3	Pesquisa	5	NULL	1988-05-22
4	Tecn Informação	6	NULL	2020-04-29

Ano-Mês-Dia mas queríamos
Ano-Dia-Mês

Ano-Mês-Dia

Observe como as datas estão ambíguas nas duas tabelas. O Sql Server espera datas **sempre** com o mês antes do dia (MM-DD, mês-dia) e as apresenta como Ano-Mês-Dia, ou seja, quando usamos

'09/01/1965' no Insert de Empregados, o servidor considerou que 09 era o mês e 01 era o dia de nascimento, ao invés de ser o dia 09 de janeiro, considerou que era 01 de setembro.

Já na tabela de Departamentos, a data está correta, pois não existem meses 19, 22 ou 29. O que fizemos de diferente é que na inclusão de Departamentos, usamos a função Convert(date) para indicar que as datas que apresentamos estão no formato 103, que é um código que o Sql Server usa para entender datas no formato DD/MM/AAAA, que é o que usamos aqui no Brasil.

Já no comando Insert de Empregados, não usamos a função Convert(date) e isso fez com que o Sql Server considerasse a data no formato padrão que ele usa, que é MM-DD-AAAA. Assim, as datas de nascimento de nossos empregados estão erradas e devemos apagar esses registros e incluí-los novamente, mas usando a função Convert(date).

Para excluir os empregados, execute o comando

```
Delete from empresa.EMPREGADO;
```

Depois disso, vamos reincluir nossos empregados e também os demais que usaremos em nosso estudo, com o comando Insert abaixo:

```
INSERT INTO Empresa.EMPREGADO
(PreNome, InicialMeio, Sobrenome, NumSegSocial, DataNascimento,
Endereco, Sexo, Salario, NumDept)
VALUES
('John', 'B', 'Smith', '123456789', Convert(date, '09/01/1965', 103),
'731 Fondren, Houston, TX', 'M', 30000, 5),
('Franklin', 'T', 'Wong', '333445555', Convert(date, '08/12/1955', 103),
'638 Voss, Houston, TX', 'M', 40000, 5),
('Alicia', 'J', 'Zelaya', '999887777', Convert(date, '19/01/1968', 103),
'3321 Castle, Spring, TX', 'F', 25000, 4),
('Jennifer', 'S', 'Wallace', '987654321', Convert(date, '20/06/1941', 103),
'291 Berry, Bellaire, TX', 'F', 43000, 4),
('Ramesh', 'K', 'Narayan', '666884444', Convert(date, '15/09/1962', 103),
'975 Fire Oak, Humble, TX', 'M', 38000, 5),
('Joyce', 'A', 'English', '453453453', Convert(date, '31/07/1972', 103),
'5631 Rice, Houston, TX', 'F', 25000, 5),
('Ahmad', 'V', 'Jabbar', '987987987', Convert(date, '29/03/1969', 103),
'980 Dallas, Houston, TX', 'M', 25000, 4),
('James', 'E', 'Borg', '888665555', Convert(date, '10/11/1937', 103),
'450 Stone, Houston, TX', 'M', 55000, 1),
('Luis', 'F', 'Assis', '512851285', Convert(date, '21/06/1994', 103),
'37 Reg Feijó, Campinas, SP', 'M', 20000, 6),
('Igor', 'N', 'Camargo', '494549454', Convert(date, '26/10/1985', 103),
'53 Dona Libânia, Campinas, SP', 'M', 30000, 6);
```

E nossa tabela passará a ter o conteúdo abaixo (Select * from Empresa.Empregado):

	PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNascimento	Endereco	Sexo	Salario	Super_n...	NumDept
1	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	NULL	5
2	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.00	NULL	5
3	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.00	NULL	5
4	Igor	N	Camargo	494549454	1985-10-26	53 Dona Libânia, Campinas, SP	M	30000.00	NULL	6
5	Luis	F	Assis	512851285	1994-06-21	37 Reg Feijó, Campinas, SP	M	20000.00	NULL	6
6	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000.00	NULL	5
7	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.00	NULL	1
8	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.00	NULL	4
9	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.00	NULL	4
10	Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000.00	NULL	4

Veja que o Sql Server dispôs os dados dessa tabela em **ordem crescente** da chave primária (NumSegSocial) e que o campo Super_numSegSocial está com null em todos os registros. Observe também que as datas de nascimento dos dois primeiros empregados tiveram o dia e o mês trocados

entre si, significando, agora, 9 de janeiro e 8 de dezembro, e não 1 de setembro e 12 de agosto como eram antes de usarmos a função Convert(date) no Insert.

O comando abaixo produzirá o resultado ao lado, onde podemos ver a separação do dia e do mês da data de nascimento através da chamada das funções day() e month() do Sql:

```
select
    prenome, day(dataNascimento) as dia,
    month(dataNascimento) as mes
from
    Empresa.Empregado
```

A necessidade de Convert(date) ocorrerá sempre que houver, no servidor de banco de dados, configurações regionais diferentes daquela do país onde os dados são gerados. Outros sistemas gerenciadores de bancos de dados poderão usar soluções diferentes para esse problema.

	prenome	dia	mes
1	John	9	1
2	Franklin	8	12
3	Joyce	31	7
4	Igor	26	10
5	Luis	21	6
6	Ramesh	15	9
7	James	10	11
8	Jennifer	20	6
9	Ahmad	29	3
10	Alicia	19	1

4.5.2. Alterando dados de tabelas

Para podermos registrar quem é o supervisor de cada empregado, precisaremos alterar os dados da tabela Empregado.

Para alterar um registro de uma tabela, usamos o comando UPDATE do SQL.

A forma geral desse comando é:

```
UPDATE nomeDaTabela
Set
    campo1 = novoValor1 [, campo2 = novoValor2 ...]
[
    Where <condição de seleção>
]
```

Por exemplo, vamos atualizar o campo Super_numSegSocial com '333445555', como consta da [figura 4.7](#):

```
UPDATE Empresa.Empregado
Set
    super_numSegSocial = '333445555'
```

A execução resulta na mensagem da figura ao lado, que diz que 10 linhas foram afetadas. O resultado vem abaixo:

UPDATE Empresa.Empregado
Set super_numSegSocial = '333445555'
(10 linhas afetadas)

PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNasci...	Endereco	Sexo	Salario	Super_numS...	NumDe...
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.00	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.00	333445555	5
Igor	N	Camargo	494549454	1985-10-26	53 Dona Libânia,Campinas,...	M	30000.00	333445555	6
Luis	F	Assis	512851285	1994-06-21	37 Reg Feijó, Campinas, SP	M	20000.00	333445555	6
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000.00	333445555	5
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.00	333445555	1
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.00	333445555	4
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.00	333445555	4
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000.00	333445555	4

Como podemos observar, todos os registros ficaram com o mesmo valor no campo Super_numSegSocial. Claramente isso não era o desejado, mas foi exatamente o que mandamos

o servidor de Banco de Dados fazer. Não dissemos qual registro específico deveria ser alterado, de forma que o comando UPDATE foi realizado sobre **todos os registros** da tabela.

Para selecionarmos qual(is) registro(s) deve(m) ser afetado(s) pelo UPDATE, temos que usar a **cláusula de seleção**, que vai **filtrar** registros de acordo com uma **condição**. Essa é a cláusula **WHERE** e ela é opcional: quando não a usamos, todos os registros serão atualizados de acordo com o ajuste definido em SET.

Para selecionarmos apenas os registros desejados, teremos de criar vários comandos UPDATE, um para cada registro desejado. Mas note na [figura 4.7](#) que os registros de Ramash Narayan e Joyce English já possuíam o valor '**333445555**' como Super_numSegSocial. Portanto, não precisaremos corrigir o Super_numSegSocial desses dois registros nem para o primeiro de todos.

Vendo na [figura 4.7](#) quais são os empregados supervisionados por cada um dos valores do campo Super_numSegSocial, podemos digitar e executar os comandos abaixo para corrigir esse campo:

```
UPDATE Empresa.Empregado
Set
    super_numSegSocial = '888665555'
where
    NumSegSocial in ('333445555', '987654321', '494549454', '512851285')

UPDATE Empresa.Empregado
Set
    super_numSegSocial = '987654321'
where
    NumSegSocial in ('999887777', '987987987')

UPDATE Empresa.Empregado
Set
    super_numSegSocial = null
where
    NumSegSocial = '888665555'
```

O resultado pode ser visualizado abaixo, com o comando `Select * from Empresa.Empregado`. Observe que James E Borg não tem nenhum supervisor porque, provavelmente, é o dono dessa companhia.

	PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNascimento	Endereco	Sexo	Salario	Super_num...	NumDe
1	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	333445555	5
2	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.00	888665555	5
3	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.00	333445555	5
4	Igor	N	Camargo	494549454	1985-10-26	53 Dona Libânia, Campinas, SP	M	30000.00	888665555	6
5	Luis	F	Assis	512851285	1994-06-21	37 Reg Feijó, Campinas, SP	M	20000.00	888665555	6
6	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000.00	333445555	5
7	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.00	NULL	1
8	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.00	888665555	4
9	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.00	987654321	4
10	Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000.00	987654321	4

Vamos agora alterar os dados da tabela DEPARTAMENTO, para que possamos indicar quem é o gerente de cada departamento cadastrado.

```
update Empresa.DEPARTAMENTO
set
    Gerente_numSegSocial = '333445555'
where numDept = 5
update Empresa.DEPARTAMENTO
set
```

```

Gerente_numSegSocial = '987654321'
where
numDept = 4

update Empresa.DEPARTAMENTO
set
    Gerente_numSegSocial = '888665555'
where
    numDept = 1

update Empresa.DEPARTAMENTO
set
    Gerente_numSegSocial = '512851285'
where
    numDept = 6

```

Esse é o resultado para a tabela Departamento:

Chamamos a função Convert(varchar) para converter a data em string e no formato dd/mm/aaaa (103)

```

select
    nomeDept, numDept, gerente_numSegSocial,
    Convert(varchar, gerente_dataInicial, 103) as Início
from
    Empresa.Departamento

```

	nomeDept	numDept	gerente_numSegSocial	Início
1	Sede	1	888665555	19/06/1981
2	Administração	4	987654321	01/01/1995
3	Pesquisa	5	333445555	22/05/1988
4	Tecn Informação	6	512851285	29/04/2020

Na página <https://www.mssqltips.com/sqlservertip/1145/date-and-time-conversions-using-sql-server/> podemos encontrar uma tabela de códigos de conversão para diversos formatos de data. Assim, você poderá configurar os comandos que buscam/exibem ou inserem datas no formato mais adequado à aplicação que estiver desenvolvendo.

4.5.3. Apagando Registros de uma tabela

Para apagar um registro de uma tabela, usamos o comando **DELETE**, cuja forma geral é:

```
DELETE FROM nomeDaTabela [ WHERE <condição de seleção> ]
```

Embora a cláusula **Where** seja opcional, seu uso é importante, pelo mesmo motivo que vimos no comando **Update**. Por exemplo:

```
Delete from Empresa.Departamento where numDept = 6
```

```
Delete from Empresa.Empregado where Sexo = 'M' and Year(DataNascimento) >= 1999
```

Obviamente, esse comando deve ser usado com muito cuidado. A **falta** da cláusula **where** fará com que **todos** os registros da tabela sejam **apagados**, a menos que restrições de integridade referencial impeça a remoção de algum registro referenciado por uma outra tabela através de uma chave estrangeira.

Caso uma tabela tenha sido configurada com a cláusula **ON DELETE CASCADE**, ela poderá ter registros apagados caso eles referenciem, via chave estrangeira, a chave primária de um registro apagado em uma outra tabela. Estudaremos isso adiante.

Lembrando, o servidor de banco de dados normalmente impede a execução de operações que violem a integridade referencial, como vimos quando tentamos incluir registros de uma tabela (fraca) que referenciavam registros de outra tabela (forte). Esse tratamento também ocorre nas operações de alteração e de exclusão, justamente para evitar as anomalias, que estudamos anteriormente.

4.5.4. Incluindo mais dados nas nossas tabelas

Inclua, agora, dados para as tabelas DEPENDENTE, DEPTO_LOCAIS, PROJETO e TRABALHA_EM, conforme os dados que vemos na [figura 4.7](#), mas com pequenas mudanças.

Abaixo temos os comandos que fazem isso, de acordo com essa figura.

```

insert
  into Empresa.DEPENDENTE
    (NumSegSocial , NomeDependente, Sexo, DataNascimento, Relacionamento)
  values
    ('333445555','Alice',      'F', Convert(date,'05/04/1986', 103), 'Filha'),
    ('333445555','Franklin',   'M', Convert(date,'25/10/1983', 103), 'Filho'),
    ('333445555','Joy',        'F', Convert(date,'03/05/1958', 103), 'Esposa'),
    ('987654321','Abner',      'M', Convert(date,'28/02/1942', 103), 'Esposo'),
    ('123456789','John',       'M', Convert(date,'04/01/1988', 103), 'Filho'),
    ('123456789','Alice',      'F', Convert(date,'30/12/1988', 103), 'Filha'),
    ('123456789','Elizabeth', 'F', Convert(date,'05/05/1967', 103), 'Esposa')

insert
  into Empresa.DEPTO_LOCAIS (NumDept, LocalDept)
  values (1, 'Houston'),
         (4, 'Stafford'),
         (5, 'Belaire'),
         (5, 'Sugarland'),
         (5, 'Houston')

insert
  into Empresa.PROJETO (NomeProjeto, NumProjeto, LocalProjeto, NumDept)
  values ('ProdutoX',          1, 'Bellaire', 5),
         ('ProdutoY',          2, 'Sugarland', 5),
         ('ProdutoZ',          3, 'Houston', 5),
         ('Informatização', 10, 'Stafford', 4),
         ('Reorganização', 20, 'Houston', 1),
         ('Novos Benefíc', 30, 'Stafford', 4)

insert
  into Empresa.TRABALHA_EM (NumSegSocial , NumProjeto, Horas)
  values ('123456789', 1, 32.5),
         ('123456789', 2, 7.5),
         ('666884444', 3, 40.0),
         ('453453453', 1, 20.0),
         ('453453453', 2, 20.0),
         ('333445555', 2, 10.0),
         ('333445555', 3, 10.0),
         ('333445555', 10, 10.0),
         ('333445555', 20, 10.0),
         ('999887777', 30, 30.0),
         ('999887777', 10, 10.0),
         ('987987987', 10, 35.0),
         ('987987987', 30, 5.0),
         ('987654321', 30, 20.0),
         ('987654321', 20, 15.0),

```

('888665555' , 20, 40.0) -- era originalmente null. Veja figura 4.7

Note que, no **Insert** acima, o último dos registros não pôde ser incluído, devido a não aceitar nulos. Nesse caso, foi arbitrado o valor de 40 horas de trabalho no projeto para não ter nulos. Esse empregado é, possivelmente, o dono da empresa e trabalha no projeto *Reorganização*, assim é justo que ele dê o exemplo trabalhando bastante (brincadeira).

Pratique as inclusões abaixo na tabela EMPREGADO, e observe os erros que aparecem:

- <'Cecilia', 'F', 'Kolonsky', NULL, Convert(date, '04-05-1960',103), '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4>
- <'Alicia', 'J', 'Zelaya', '999887777', Convert(date, '04-05-1960',103), '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4>
- <'Cecilia', 'F', 'Kolonsky', '677678989', Convert(date, '04-05-1960',103), '6357 Windswept, Katy, TX', F, 28000, '987654321', 7>
- <'Cecilia', 'F', 'Kolonsky', '677678989', Convert(date, '04-05-1960',103), '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4>

A seguir vemos uma explicação dos erros ocorridos:

- Esta inserção viola a restrição de integridade da tabela (NULL para a chave primária NumSegSocial) e, assim, o comando foi rejeitado.
- Esta inserção viola a restrição de chave porque outro registro com o mesmo valor de NumSegSocial já existe na tabela EMPREGADO e, por isso, o comando foi rejeitado.
- Esta inserção viola a restrição de integridade referencial especificada em NumDept na tabela EMPREGADO, porque não há um registro referenciado correspondente na tabela DEPARTAMENTO com NumDept = 7.
- Esta inserção satisfaz todas as restrições, de forma que o comando foi aceito, executado e o registro inserido.

Como prática, crie também um Dependente para o funcionário Igor e adicione registros para esse funcionário e para o funcionário Luis na tabela TRABALHA_EM.

[Vídeo adicional – inclusão de dados em tabelas](#)

[Vídeo adicional – alteração de dados em tabelas](#)

4.5.5. Interação dos Comandos de Manipulação de Dados com Constraints da tabela

Quando criamos tabelas, podemos definir restrições aos campos, incluindo restrições de chave, de integridade referencial, valores default, valores aceitáveis e não-nulos.

A cláusula DEFAULT permite definir um valor padrão para preenchimento de um campo quando, no comando INSERT, não é explicitado um valor para esse campo. Por exemplo, para as tabelas EMPREGADO e DEPARTAMENTO:

```
CREATE TABLE EMPREGADO
( . . .
  numSegSocial char(9) not null,
  NumDepto INT NOT NULL DEFAULT 1,
  Super_numSegSocial char(9),
  CONSTRAINT EMPPK
    PRIMARY KEY (NumSegSocial),
  CONSTRAINT EMPSUPERFK
    FOREIGN KEY (Super_numSegSocial) REFERENCES EMPREGADO
      (NumSegSocial)
  CONSTRAINT EMPDEPTFK
    FOREIGN KEY(NumDepto) REFERENCES DEPARTAMENTO(NumDepto)
    ON DELETE SET DEFAULT
    ON UPDATE CASCADE
);
```

DEFAULT informa o valor que o campo receberá na inclusão caso esse valor não seja explicitamente informado no Insert

Aqui se atribuiu um nome identificador para as chaves primária e estrangeiras

Tratamento de violações de restrições de integridade referencial

A ação default que SQL toma para uma violação de integridade é **rejeitar** a operação de atualização que causaria a violação, o que é conhecido como abordagem RESTRITA.

No entanto, o projetista pode especificar uma ação alternativa a ser tomada, pelo uso de uma cláusula de **ação referencial disparada** a qualquer constraint de Foreign Key. As opções incluem SET NULL, CASCADE e SET DEFAULT.

Uma opção deve ser qualificada com tanto ON DELETE ou ON UPDATE.

Em geral, a ação tomada pelo Gerenciador de Banco de Dados para SET NULL ou SET DEFAULT é a mesma tanto para ON DELETE quanto para UPDATE: o valor dos atributos referenciadores é alterado para NULL com o SET NULL e para o valor default especificado para o atributo referenciado com SET DEFAULT.

A ação de CASCADE ON DELETE é remover todos os registros referenciadores; já a ação para CASCADE ON UPDATE é alterar o valor do atributo chave estrangeira referenciador para o novo valor da chave primária para os registros que a referenciam.

É responsabilidade do projetista do banco de dados escolher a ação apropriada e especificá-la.

Como uma regra geral, a opção CASCADE é adequada para tabelas em relacionamentos tais como TRABALHA_EM, para tabelas que representam atributos multivvalorados, tais como DEPTO_LOCAIS e para tabelas que representam entidades fracas, tais como DEPENDENTE.

Nos comandos acima, o projetista do banco de dados decidiu não colocar ON DELETE nem ON UPDATE na declaração da chave estrangeira EMPSUPERFK (que autorreferencia a tabela pelo campo NumSegSocial). Essa decisão decorre do fato que Sql Server **impede** isso em **autorrelacionamentos** para evitar uma situação de corrida hierárquica em remoções e atualizações. Lembre-se que ai temos uma hierarquia entre supervisor e supervisionado e o SQL Server simplesmente impede que remoções ou atualizações em cascata possam acontecer nesse caso, por questão de segurança e evitar dificuldades na análise de quem é subordinado a quem.

Mas veja a cláusula abaixo, da tabela EMPREGADO:

```

CONSTRAINT EMPDEPTFK
    FOREIGN KEY(NumDept) REFERENCES DEPARTAMENTO(NumDept)
    ON DELETE SET DEFAULT
    ON UPDATE CASCADE

```

Elá indica que, se um registro qualquer de DEPARTAMENTO for apagado, acontecerão as seguintes ações referenciais na tabela EMPREGADO:

Quando se apagar um registro de DEPARTAMENTO: o campo EMPREGADO.NumDept receberá um valor default (no caso, 1) se tiver o mesmo valor que o DEPARTAMENTO.NumDept que foi apagado

Quando se alterar o valor de DEPARTAMENTO.NumDept: todos os registros de EMPREGADO onde o campo EMPREGADO.NumDept seja igual ao campo DEPARTAMENTO.NumDept original **serão atualizados para o novo valor** de DEPARTAMENTO.NumDept.

Outros tipos de constraints

Analise, agora, o comando abaixo e veja o que fazem as cláusulas Check e Unique:

```

CREATE TABLE DEPARTAMENTO
(
    NumDept int not null,
        CHECK (NumDept > 0 and NumDept < 21),
    Gerente_numSegSocial CHAR(9) NOT NULL DEFAULT '888665555',
    NomeDept varchar(15),
    Data_CriacaoDept Date,
    Gerente_dataInicial Date
        CHECK (Data_CriacaoDept <= gerente_DataInicial),
    CONSTRAINT DEPTPK PRIMARY KEY(NumDept),
    CONSTRAINT DEPTSK UNIQUE(NomeDept),
    CONSTRAINT DEPTMGRFK
        FOREIGN KEY (Gerente_numSegSocial)
            REFERENCES EMPREGADO (NumSegSocial)
            ON DELETE SET DEFAULT
            ON UPDATE CASCADE
);

```

The diagram shows the `CREATE TABLE DEPARTAMENTO` command with three callout boxes explaining specific clauses:

- Annotation 1 (for `CHECK (NumDept > 0 and NumDept < 21)`):** CHECK faz com que, quando esse campo receber um valor, em inclusão ou alteração, seja verificado se o valor recebido atende à condição especificada.
- Annotation 2 (for `CONSTRAINT DEPTSK UNIQUE(NomeDept)`):** CHECK também funciona para comparação de campos distintos no mesmo registro em inclusões e alterações de registros.
- Annotation 3 (for `FOREIGN KEY (Gerente_numSegSocial)`):** UNIQUE evita repetição de valor desse campo, mesmo não sendo uma chave primária.

No comando acima, além de **CHECK** e **UNIQUE**, temos também a declaração de ações referenciais, que funcionarão como descrito a seguir:

No caso de um registro de EMPREGADO ser apagado, todos os registros de DEPARTAMENTO onde Gerente_numSegSocial seja igual ao NumSegSocial do registro apagado terão o campo Gerente_numSegSocial alterado para o valor Default ('888665555').

Já caso aconteça a alteração de um EMPREGADO.NumSegSocial, todos os registros de DEPARTAMENTO cujo campo Gerente_numSegSocial seja igual a esse NumSegSocial serão alterados para o novo valor.

4.6. Consulta a Dados

SQL tem um único comando para recuperar (consultar) informações de um Banco de Dados. É o comando **SELECT**. Ele é bastante poderoso e, por isso, estudaremos primeiramente suas opções mais simples.

4.6.1. Formato e uso básicos do comando Select

```
Select <lista de atributos> → campos que devem ser recuperados pela consulta
from <lista de tabelas> → lista de tabelas que serão pesquisadas na consulta
[ where <condição>; ] → esta é a condição que identifica os registros desejados
```

Na <condição>, para comparar valores usamos os operadores relacionais <. <=. >=, = e <>. Os dois últimos são usados para operações de igualdade e de desigualdade, como ocorre na linguagem Delphi e diferentemente do que usamos em C#, Java e Javascript. Usamos também os operadores lógicos AND e OR para trabalhar com condições compostas e NOT para inverter o resultado de uma condição. Outros operadores serão apresentados gradualmente no decorrer de nosso estudo.

Em seguida praticaremos algumas consultas de amostra em nosso banco de dados de estudo.

O comando SELECT da SQL especifica os atributos cujos valores devem ser recuperados, os quais são chamados de **atributos de projeção**, e a cláusula WHERE especifica a condição lógica que deve ser verdadeira para quaisquer dos registros recuperados, condição essa conhecida por **condição de seleção**.

Consulta 0: recupera a data de nascimento e endereço do(s) empregado(s) cujo nome é ‘John B. Smith’:

```
SELECT convert(varchar, DataNascimento, 103) Nascimento,
       Endereco
  FROM Empresa.EMPREGADO
 WHERE PreNome='John' AND InicialMeio='B' AND Sobrenome='Smith';
```

Os campos retornados são
chamados de **Projeção** na
teoria relacional

Esta é a condição que indica os **critérios de seleção** para satisfazer a consulta

Esta consulta envolve somente uma tabela, a EMPREGADO, listada na cláusula FROM do comando SELECT. Essa consulta seleciona, na tabela gravada no banco de dados, os registros que satisfazem os **critérios de seleção** informados na condição. Dentre esses registros, os dados das colunas DataNascimento e Endereco são **projetados** no resultado que será apresentado ao usuário que fez a consulta.

Podemos imaginar que existe uma variável de registro implícita na consulta SQL que percorrerá cada registro individual da tabela EMPREGADO e avaliará se os campos desse registro atendem à condição de seleção apresentada na cláusula WHERE. Somente os registros que atendam a essa condição serão selecionados, ou seja, aqueles registros em que a condição resulte em TRUE ao ser aplicada aos campos desse registro. No caso do nosso banco de dados, o resultado da seleção de registros seria esse:

PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNascimento	Endereco	Sexo	Salario	Super_numSegSocial	NumDept
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	333445555	5

Sobre os registros selecionados acima é feita a operação de **projeção**, que cria um conjunto de linhas contendo apenas os dados das colunas (atributos) especificados após a palavra SELECT:

PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNascimento	Endereco	Sexo	Salario	Super_numSegSocial	NumDept
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	333445555	5

E o resultado final, retornado ao computador cliente, seria como abaixo:

	Nascimento	Endereco
1	09/01/1965	731 Fondren, Houston, TX

Consulta 1: recupera o nome e o endereço de todos os empregados que trabalham no departamento ‘Pesquisa’:

```
SELECT Prenome, Sobrenome, Endereco
FROM
    Empresa.EMPREGADO,
    Empresa.DEPARTAMENTO
WHERE
    NomeDept = 'Pesquisa' AND NumDept = NumDept;
```

Na cláusula WHERE da consulta acima, a condição NomeDept = ‘Pesquisa’ é a **condição de seleção** que escolhe o registro de interesse específico na tabela DEPARTAMENTO, porque NomeDept é um campo de DEPARTAMENTO.

A condição NumDept = NumDept é chamada de **condição de junção**, pois ela combina dois registros: um da tabela DEPARTAMENTO e um da tabela EMPREGADO, sempre que o valor de NumDept em DEPARTAMENTO é igual ao valor de NumDept em EMPREGADO. Mas como o servidor saberá em qual tabela o campo numDept se encontra, pois usamos o mesmo nome para esse campo, nas duas tabelas? Essa é uma decisão de modelagem; o uso de nomes iguais para campos que significam a mesma coisa pode ser considerado adequado, em termos de dicionário de dados e da criação do vocabulário necessário para o desenvolvimento do projeto.

No entanto, aqui acaba criando uma ambiguidade e precisamos resolvê-la, pois o próprio servidor já mostra um erro no momento em que digitamos isso na janela de consulta do SSMS:

```
WHERE
NomeDept='Pesquisa' AND NumDept=NumDept;
```

Ambiguous column name 'NumDept'.

Uma maneira de resolver a ambiguidade é informar os nomes das tabelas antes dos campos, usando a operação chamada Qualificação de Campo, como vemos abaixo:

```
WHERE
NomeDept = 'Pesquisa' AND Empregado.NumDept = Departamento.NumDept;
```

Em geral, qualquer número de condições de seleção e de junção pode ser especificado em uma única consulta SQL. Abaixo vemos o resultado dessa consulta.

Observe os dados na [figura 4.7](#) e note a junção de informações das duas tabelas.

	PreNome	Sobrenome	Endereco
1	John	Smith	731 Fondren, Houston, TX
2	Franklin	Wong	638 Voss, Houston, TX
3	Joyce	English	5631 Rice, Houston, TX
4	Ramesh	Narayan	975 Fire Oak, Humble, TX

A figura a seguir mostra, em verde, a seleção do registro de DEPARTAMENTO cujo nome é igual a ‘Pesquisa’, cujo valor de NumDept é 5. Em seguida, podemos ver igualdade dos campos Empregado.NumDept e Departamento.NumDept entre as duas tabelas (valendo 5). Os registros de EMPREGADO que atendem a essa igualdade são selecionados e, sobre eles, é aplicada a projeção nos campos prenome, sobrenome e endereço:

DEPARTAMENTO

NomeDept	NumDept	Gerente_numSegSocial	Gerente_dataNicial
Sede	1	888665555	1981-06-19
Administração	4	987654321	1995-01-01
Pesquisa	5	333445555	1988-05-22
Tecn Informação	6	512851285	2020-04-29

EMPREGADO

PreNome	IncialMeio	Sobrenome	NumSe...	Data...	Endereco	Sexo	Salario	Super_...	NumDept
John	B	Smith	12345...	1965...	731 Fondren, Houston, TX	M	30000.00	333445...	5
Franklin	T	Wong	33344...	1955...	638 Voss, Houston, TX	M	40000.00	888665...	5
Joyce	A	English	45345...	1972...	5631 Rice, Houston, TX	F	25000.00	333445...	5
Igor	N	Camargo	49454...	1985...	53 Dona Libânia,Campinas,SP	M	30000.00	888665...	6
Luis	F	Assis	51285...	1994...	37 Reg Feijó, Campinas, SP	M	20000.00	888665...	6
Ramesh	K	Narayan	66688...	1962...	975 Fire Oak, Humble, TX	M	38000.00	333445...	5
Cecilia	F	Kolonsky	67767...	1960...	6357 Windy Lane, Katy, TX	F	28000.00	NULL	4
James	E	Borg	88866...	1937...	450 Stone, Houston, TX	M	55000.00	NULL	1
Jennifer	S	Wallace	98765...	1941...	291 Berry, Bellaire, TX	F	43000.00	888665...	4
Ahmad	V	Jabbar	98798...	1969...	980 Dallas, Houston, TX	M	25000.00	987654...	4
Alicia	J	Zelaya	99988...	1968...	3321 Castle, Spring, TX	F	25000.00	987654...	4

Uma consulta que envolva somente seleção e condições de junção mais atributos de projeção é chamada de consulta **seleção-projeção-junção**. O próximo exemplo é uma consulta seleção-projeção-junção com *duas* condições de junção.

Consulta 2: Para cada projeto localizado em ‘Stafford’, lista o número do projeto, o número do departamento que controla esse projeto e o sobrenome, endereço e data de nascimento do gerente do departamento:

SELECT

```
NumProjeto, Departamento.NumDept, Sobrenome, Endereco, DataNascimento
FROM
    Empresa.PROJETO, Empresa.DEPARTAMENTO, Empresa.EMPREGADO
WHERE
    Projeto.NumDept = Departamento.NumDept AND
    Gerente_numSegSocial = NumSegSocial AND
    LocalProjeto='Stafford';
```

A condição de junção Projeto.NumDept = Departamento.NumDept relaciona um registro de projeto ao registro do seu departamento controlador, enquanto a condição de junção Gerente_numSegSocial = NumSegSocial relaciona o departamento controlador do projeto aos registros de empregados que gerenciam esse departamento. Cada registro no conjunto de resultados (**result set**) será uma **combinação** de um projeto, um departamento e um empregado que satisfaz as condições de junção. Os atributos de projeção são usados para escolher os atributos que serão apresentados em cada combinação de registros.

A figura a seguir apresenta as tabelas com os dados originais e a sequência de operações realizadas para buscar os dados desejados. As cores na figura indicam a ordem das operações:

1. Verde
2. Azul
3. Vermelho
4. Laranja

PROJETO

NomeProjeto	NumProjeto	LocalProjeto	NumDept
ProdutoX	1	Bellaire	5
ProdutoY	2	Sugarland	5
ProdutoZ	3	Houston	5
Informatização	10	Stafford	4
Reorganização	20	Houston	1
Novos Benefíc	30	Stafford	4

Primeiramente, buscamos os registros onde LocalProjeto sejam iguais a Stafford. Como LocalProjeto é um campo da tabela PROJETO, é nessa tabela que a consulta se inicia. Vemos dois registros com LocalProjeto igual a Stafford (marcados em **verde**).

Nesses registros, o NumDept é igual a 4 (em **azul**).

DEPARTAMENTO

NomeDept	NumDept	Gerente_numSegSocial
Sede	1	888665555
Administração	4	987654321
Pesquisa	5	333445555
Tecn Inform...	6	512851285

Daí, na tabela de Departamento procuramos registros onde Departamento.NumDept seja igual a Projeto.NumDept (igual a 4). Encontramos o registro onde NomeDept é igual a Administração, tendo NumDept igual a 4. Nesse registro também temos o campo Gerente_numSegSocial que será usado na próxima condição, e o valor desse campo nesse registro está marcado em **vermelho** ('987654321').

EMPREGADO

PreNome	Inic...	Sobrenome	NumSegSocial	DataNasci...	Endereco	S...	Sala...	Super...	NumDept
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	300...	3334...	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	400...	8886...	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	250...	3334...	5
Igor	N	Camargo	494549454	1985-10-26	53 Dona Libânia,Campinas,SP	M	300...	8886...	6
Luis	F	Assis	512851285	1994-06-21	37 Reg Feijó, Campinas, SP	M	200...	8886...	6
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	380...	3334...	5
Cecilia	F	Kolonsky	677678989	1960-04-05	6357 Windy Lane, Katy, TX	F	280...	NULL	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	550...	NULL	1
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	430...	8886...	4
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	250...	9876...	4
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	250...	9876...	4

Esse valor ('987654321') será comparado com o campo NumSegSocial, buscando igualdades. Como esse campo é da tabela EMPREGADO, e não é usado em nenhuma das outras tabelas da consulta, é nessa tabela que buscaremos um registro que tenha esse valor no campo NumSegSocial.

A partir daí, basta **juntar** os registros (marcados em **laranja**) e fazer a **projeção** dos campos, para termos os resultados abaixo:

Resultados		Mensagens			
	NumProjeto	NumDept	Sobrenome	Endereco	DataNascimento
1	10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
2	30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

Caso deseje que a data de nascimento seja apresentada no formato que usamos no Brasil, basta chamar a função Convert(varchar) como fizemos anteriormente.

Note que, se você estiver fazendo essa consulta a partir de um programa que desenvolveu, e não do SSMS, o próprio programa poderá formatar o campo DataNascimento quando este for retornado, pois um campo de tipo Date é, em geral, comprehensível em todas as linguagens de programação.

4.6.2. Nomes ambíguos de atributos, Apelidos, Variáveis de Registros

Nas explicações acima, observamos que o SQL busca, a partir dos nomes dos campos da lista de atributos, quais tabelas deverão ser pesquisadas. Mas, caso haja nomes de campos iguais em tabelas distintas, como já tivemos de lidar? Isso é possível e, muitas vezes, é até recomendável, para que a **chave primária de uma tabela tenha o mesmo nome** de campo que a **chave estrangeira da tabela associada num relacionamento**. Dessa forma, é mantida uma coerência na nomenclatura dos campos. Mas não somente nesse caso de chaves primária e estrangeiras pode haver repetição de nomes de campos. Por exemplo, a palavra **nome** pode ser usada para os atributos *nome de funcionário*, *nome de departamento* e *nome de projeto*. Seria até mesmo uma boa escolha para o nome desses campos.

Quando isso acontece, usamos o que chamamos de **ALIAS** ou **Apelido** da tabela para diferenciar um campo de outro com o mesmo nome. Essa operação se chama de **qualificação**. Colocamos o nome da tabela antes do nome do campo e um ponto entre os dois. Por exemplo:

Empregado.name

Departamento.name

Projeto.name

Imagine que na tabela Empregado, o campo Sobrenome fosse chamado de name, e o campo nomeDept da tabela Departamento também se chamassem name. Então, para prevenir ambiguidade, a consulta 1 deveria ser refeita como mostrado abaixo. Devemos prefixar os atributos name, especificando a quais tabelas nos referimos, porque esses nomes são repetidos nas duas tabelas:

Consulta 1a: recupera o nome e o endereço de todos os empregados que trabalham no departamento cujo nome é ‘Pesquisa’:

```
SELECT prenome, EMPREGADO.name, endereco
FROM EMPREGADO,
DEPARTAMENTO
WHERE
DEPARTAMENTO.name = 'Pesquisa' AND
DEPARTAMENTO.NumDept = EMPREGADO.NumDept;
```

Mesmo na consulta 1 original usamos os nomes das tabelas como prefixos aos campos usados no comando Select, para que não houvesse dúvidas sobre quais campos estão sendo usados. De fato, essa prática é recomendada porque ela torna as consultas mais fáceis de serem compreendidas.

Também podemos usar um **Alias** ou Apelido da tabela, como vemos na **Consulta 1b** abaixo:

```
SELECT prenome, Emp.name, endereco
FROM Empresa.EMPREGADO as Emp,
Empresa.DEPARTAMENTO as D
WHERE
D.name = 'Pesquisa' AND
D.NumDept = Emp.NumDept;
```

Usando os nomes dos campos de nossas tabelas, essa consulta poderia ser escrita como:

```
SELECT prenome, sobrenome, endereco
FROM Empresa.EMPREGADO as Emp,
Empresa.DEPARTAMENTO as D
WHERE
NomeDept= 'Pesquisa' AND
D.NumDept = Emp.NumDept;
```

A ambiguidade de nomes de atributos também aparece no caso em que as consultas se referem à mesma tabela duas vezes, como no exemplo abaixo:

Consulta 3: para cada empregado, recuperar o prenome e sobrenome do empregado e o prenome e sobrenome do seu supervisor imediato.

```
SELECT E.PreNome, E.Sobrenome, S.PreNome, S.Sobrenome
FROM Empresa.EMPREGADO AS E,
      Empresa.EMPREGADO AS S
WHERE E.Super_numSegSocial = S.NumSegSocial;
```

Os nomes que usamos depois da palavra reservada **AS** são chamados de Alias (apelidos) ou **variáveis de Tupla**. A palavra AS é opcional, de forma que você poderia escrever na forma abaixo:

```
FROM EMPREGADO E, EMPREGADO S
```

Também é possível indicar Alias para os nomes dos campos, como vemos abaixo:

```
SELECT E.prenome as Prenome, E.sobrenome as Sobrenome,
       S.prenome NomeSup, S.sobrenome SobrenomeSup
FROM Empresa.EMPREGADO E,
      Empresa.EMPREGADO S
WHERE E.Super_numSegSocial=S.NumSegSocial;
```

O resultado dessa consulta vem ao lado. Observe os cabeçalhos de cada coluna e os compare com os apelidos dos campos usados acima:

Nas duas consultas anteriores, podemos pensar em E e S como duas cópias distintas da tabela EMPREGADO: a primeira representa os empregados no papel de supervisionados e a segunda representa empregados no papel de supervisores.

Podemos então juntar as duas cópias. Óbvio que, na realidade, há apenas uma **única** tabela EMPREGADO, e a condição de junção busca juntar a tabela consigo mesma combinando os registros que satisfaçam E.Super_numSegSocial= S.NumSegSocial.

Esse tipo de consulta, chamada de **consulta recursiva**, é permitida pelo auto-relacionamento dessa tabela, mas não podia ser feito em versões mais antigas do SQL. A partir da versão SQL:1999 esse recurso foi incorporado.

Vídeo adicional - Alias

4.6.3. Cláusula Where não-especificada e uso do Asterisco

Não se usar uma cláusula Where no comando Select indica que **não se deseja selecionar registros**, ou seja, todos os registros da relação especificada na cláusula FROM serão retornados no conjunto de resultados.

Se houver **mais de uma tabela** especificada na cláusula FROM, e **não houver Where**, ocorrerá um **Produto Cartesiano** entre as tabelas – todas as possíveis combinações de registros serão retornadas. Não é necessário haver um relacionamento entre as tabelas envolvidas. Mas é importante ter cuidado pois, se você tiver uma tabela “A” com 20.000 linhas e 40.000 linhas na tabela “B”, você terá 800.000 linhas combinadas, como resultado (algumas pessoas comentam que essa é uma boa forma de se gerar uma massa de dados para teste).

	Prenome	Sobrenome	NomeSup	SobrenomeSup
1	John	Smith	Franklin	Wong
2	Franklin	Wong	James	Borg
3	Joyce	English	Franklin	Wong
4	Igor	Camargo	James	Borg
5	Luis	Assis	James	Borg
6	Ramesh	Narayan	Franklin	Wong
7	Jennifer	Wallace	James	Borg
8	Ahmad	Jabbar	Jennifer	Wallace
9	Alicia	Zelaya	Jennifer	Wallace

Por exemplo, a consulta 4 seleciona todos os NumSegSocial de EMPREGADO e a consulta 5 seleciona todos os NomeDeptos de DEPARTAMENTO. Já a consulta 6 seleciona **todas as combinações** de um NumSegSocial de EMPREGADO e de NomeDeptos de DEPARTAMENTO, sem se importar se o empregado trabalha ou não no departamento:

Consulta 4

```
Select
    NumSegSocial
From
    Empresa.EMPREGADO
```

Resultados		Mesa
	numsegSocial	
1	123456789	
2	333445555	
3	453453453	
4	494549454	
5	512851285	
6	666884444	
7	677678989	
8	888665555	
9	987654321	
10	987987987	
11	999887777	

Consulta 5

```
Select
    NomeDeptos
From
    Empresa.DEPARTAMENTO
```

Resultados		Mesa
	nomeDeptos	
1	Administração	
2	Pesquisa	
3	Sede	
4	Tecn Informação	

Consulta 6

```
Select
    NumSegSocial,
    NomeDeptos
From
    Empresa.EMPREGADO,
    Empresa.DEPARTAMENTO
```

	numsegSocial	nomeDeptos
1	123456789	Administração
2	333445555	Administração
3	453453453	Administração
4	494549454	Administração
5	512851285	Administração
6	666884444	Administração
7	677678989	Administração
8	888665555	Administração
9	987654321	Administração
10	987987987	Administração
11	999887777	Administração
12	123456789	Pesquisa
13	333445555	Pesquisa
14	453453453	Pesquisa
15	494549454	Pesquisa
16	512851285	Pesquisa
17	666884444	Pesquisa
18	677678989	Pesquisa
19	888665555	Pesquisa
20	987654321	Pesquisa
21	987987987	Pesquisa
22	999887777	Pesquisa
23	123456789	Sede
24	333445555	Sede
25	453453453	Sede
26	494549454	Sede
27	512851285	Sede
28	666884444	Sede
29	677678989	Sede
30	888665555	Sede
31	987654321	Sede
32	987987987	Sede
33	999887777	Sede
34	123456789	Tecn Inform...
35	333445555	Tecn Inform...
36	453453453	Tecn Inform...
37	494549454	Tecn Inform...
38	512851285	Tecn Inform...
39	666884444	Tecn Inform...
40	677678989	Tecn Inform...
41	888665555	Tecn Inform...
42	987654321	Tecn Inform...
43	987987987	Tecn Inform...
44	999887777	Tecn Inform...

Podemos observar que há **onze** registros na **consulta 4** e **quatro** registros na **consulta 5**.

Já na **consulta 6**, tivemos **quarenta e quatro** registros, pois todos os registros de EMPREGADO foram combinados com todos os registros de DEPARTAMENTO.

É extremamente importante especificar cada condição de seleção e de junção na cláusula WHERE; se alguma dessas condições for esquecida, resultados incorretos e muito extensos podem ser retornados.

Caso você realmente deseje realizar um produto cartesiano entre duas tabelas, poderá usar a cláusula **CROSS JOIN** que, ao menos, deixa mais claro que se deseja cruzar todos os registros das tabelas participantes, como vemos na **Consulta 7**, abaixo:

Consulta 7:

```
Select NumSegSocial, NomeDeptos
FROM
    Empresa.EMPREGADO
CROSS JOIN
    Empresa.DEPARTAMENTO
```

O resultado desse comando é exatamente igual ao da **Consulta 6**, visto na figura ao lado.

Clique [aqui](#) para um artigo na Internet sobre Produto Cartesiano.

Para recuperar os valores de todos os campos dos registros selecionados, não precisamos escrever a lista de campos explicitamente em SQL; podemos especificar um asterisco (*), que indicará *todos os campos*.

Por exemplo, a consulta 1c recupera todos os valores de campos de qualquer EMPREGADO que trabalha no DEPARTAMENTO de número 5;

Consulta 1c:

```
SELECT *
FROM Empresa.EMPREGADO
WHERE NumDept=5;
```

	Resultados										Mensagens	
	PreNome	InicialMeio	Sobrenome	NumSegSocial	DataNascimento	Endereco	Sexo	Salario	Super_numSegSocial	NumDept		
1	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	333445555	5		
2	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.00	888665555	5		
3	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.00	333445555	5		
4	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000.00	333445555	5		

A consulta 1d recuperará todos os campos de EMPREGADO e de DEPARTAMENTO para cada empregado que trabalha no departamento ‘Pesquisa’:

Consulta 1d:

```
SELECT *
FROM Empresa.EMPREGADO E,
      Empresa.DEPARTAMENTO D
WHERE NomeDept= 'Pesquisa' AND E.NumDept=D.NumDept;
```

PreNome	InicialM...	Sobrenome	NumSegSo...	DataNascim...	Endereco	Sexo	Salario	Super_numSegS...	NumDe...	NomeDept	NumDe...	Gerente_numSeg...	Gerente_datal...
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000....	333445555	5	Pesquisa	5	333445555	1988-05-22
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000....	888665555	5	Pesquisa	5	333445555	1988-05-22
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000....	333445555	5	Pesquisa	5	333445555	1988-05-22
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000....	333445555	5	Pesquisa	5	333445555	1988-05-22

Como pedimos todos os campos, observe que o campo de junção (numDept) aparece duas vezes, uma para cada tabela que participa da condição de junção.

[Vídeo adicional – Comando Select](#)

[Vídeo adicional – Cláusula Where](#)

[Vídeo adicional – Operadores Lógicos And e Or](#)

4.6.4. Tabelas como Conjuntos

SQL geralmente trata uma tabela não como um conjunto, mas como um **multiconjunto**: linhas *duplicadas podem aparecer mais que uma vez em uma tabela e no resultado de uma consulta*.

Embora seja recomendado que toda tabela tenha uma chave primária, para que não haja repetições entre registros, isso não é obrigatório.

SQL não elimina automaticamente registros duplicados nos resultados de uma consulta, pelas seguintes razões:

- Eliminação de duplicados é uma operação custosa. Uma maneira de implementá-la é primeiramente ordenar os registros (o que toma tempo de processamento) e depois eliminar as duplicatas de registros subsequentes e iguais (mais tempo gasto);
- O usuário pode querer ver registros duplicados no resultado de uma consulta;
- Quando uma função agregadora (estudaremos adiante esse assunto) é aplicada a registros, na maioria dos casos não queremos eliminar as duplicatas.

Uma tabela com uma chave primária possui a restrição de funcionar como um conjunto, em que os elementos (registros) não são totalmente iguais (pelo menos a chave primária será diferente em cada um). Assim, os registros de uma tabela com chave primária são **distintos** entre si.

No entanto, em consultas isso não acontece, porque resultados de consultas (select) não possuem chave primária, já que não são tabelas reais e sim “virtuais”, ficam em memória apenas para leitura e não são gravadas em disco.

Se quisermos eliminar registros duplicados de um result set de uma consulta SQL, precisaremos acoplar a cláusula **DISTINCT** no comando SELECT, indicando que somente registros distintos devem compor o resultado retornado pela consulta.

Em geral, uma consulta com SELECT DISTINCT elimina duplicatas, enquanto uma consulta com SELECT ALL não o faz. Um SELECT **sem ALL nem DISTINCT** equivale a um **SELECT ALL**.

A **Consulta 8** retorna o salário de cada empregado; se vários empregados tiverem o mesmo valor de salário, esse salário aparecerá tantas vezes quanto o número de empregados que o tem.

Mas, se estivermos interessados apenas nos diferentes valores de salários aplicados pela companhia, teremos de executar a **Consulta 9**, que retorna os mesmos salários que a **Consulta 8**, mas cada valor distinto aparecerá apenas uma vez.

Consulta 8

```
select
    Salario
from
    Empresa.EMPREGADO
```

	Salario
1	30000.00
2	40000.00
3	25000.00
4	30000.00
5	20000.00
6	38000.00
7	28000.00
8	55000.00
9	43000.00
10	25000.00
11	25000.00

Cada diferente salário aparece várias vezes no result set.

Consulta 9

```
select distinct
    Salario
from
    Empresa.EMPREGADO
```

	Salario
1	20000.00
2	25000.00
3	28000.00
4	30000.00
5	38000.00
6	40000.00
7	43000.00
8	55000.00

Cada diferente salário aparece apenas uma vez no result set.

Outros tipos de operações podem ser realizados para tratar resultados de consultas como conjuntos. SQL incorporou diretamente algumas das operações de conjuntos da Teoria Matemática de Conjuntos, que também são parte da **Álgebra Relacional** (a base teórica dos bancos de dados relacionais). Essas operações são as de União de Conjuntos (operador **UNION**), Diferença de Conjuntos (operador **EXCEPT**) e a Intersecção de Conjuntos (operador **INTERSECT**).

Os resultados dessas operações são conjuntos de registros, ou seja, registros duplicados são eliminados do resultado (mas não das tabelas originais).

Essas operações de conjuntos se aplicam somente a resultados compatíveis com união, de forma que nós temos de assegurar que as duas tabelas onde aplicamos os operadores devem ter os mesmos campos e esses campos apareçam na mesma ordem em ambas as tabelas.

Vídeo Adicional – cláusula Distinct

O próximo exemplo ilustra o uso do operador **UNION**:

Consulta 10: faça uma lista de todos os números de projeto que envolvem um empregado cujo sobrenome seja ‘Smith’, tanto como um funcionário ou como gerente do departamento que controla o projeto.

```
SELECT DISTINCT numProjeto
FROM
    Empresa.PROJETO P,
    Empresa.DEPARTAMENTO D,
    Empresa.EMPREGADO E
WHERE
    P.numDept = D.numDept AND Gerente_numSegSocial = numSegSocial AND
    Sobrenome = 'Smith'
UNION
SELECT DISTINCT P.numProjeto
FROM
    Empresa.PROJETO P,
    Empresa.TRABALHA_EM T,
    Empresa.EMPREGADO E
WHERE
    P.numProjeto = T.numProjeto AND T.numSegSocial = E.numSegSocial AND
    Sobrenome = 'Smith';
```

O primeiro SELECT retorna os números de projetos que envolvem um ‘Smith’ como gerente do departamento que controla o projeto, e o segundo retorna os números de projeto que envolvem um ‘Smith’ como um funcionário no projeto. Note que se vários empregados tiverem o sobrenome ‘Smith’, os números de projetos envolvendo qualquer um deles serão retornados.

A aplicação do operador UNION aos resultados dos dois SELECTs gera o resultado completo desejado, que vemos abaixo:

Resultados	
	NumProjeto
1	1
2	2

SQL também tem os mesmos operadores (UNION, EXCEPT e INTERSECT) correspondentes para multiconjuntos, que são seguidos pela palavra reservada **ALL** (UNION ALL, EXCEPT ALL, INTERSECT ALL). Seus resultados são multiconjuntos, ou seja, registros duplicados não são eliminados.

Clique [aqui](#) para saber mais sobre **Except** e **Intersect**.

Vídeo adicional – operador Union

4.6.5. Combinação de padrões de subcadeia de pesquisa e operadores aritméticos

Vamos discutir agora alguns outros recursos do comando Select. O primeiro recurso permite condições de comparação feitas sobre somente partes de uma cadeia de caracteres, através do uso do operador de comparação **LIKE**. Ele pode ser usado para combinação de padrões de strings.

Cadeias parciais de caracteres são especificadas pelo uso de dois caracteres especiais: % e _ . % substitui um número arbitrário de zero ou mais caracteres, e o _ substitui um único caracter. Esses caracteres especiais são como *coringas*. Por exemplo, considere a consulta abaixo:

Consulta 11: retorna todos os empregados cujos endereços fiquem em Houston, Texas.

```
Select
    preNome, sobrenome
from
    Empresa.EMPREGADO
where
    Endereco like '%Houston, tx%'
```

	preNome	sobrenome
1	John	Smith
2	Franklin	Wong
3	Joyce	English
4	James	Borg
5	Ahmad	Jabbar

Vemos os resultados na figura acima. Observe que não foi preciso colocar o estado (TX) em maiúsculo pois, por padrão, o SQL Server **não diferencia maiúsculas ou minúsculas em consultas**.

Para recuperar todos os empregados nascidos durante a década de 1950, podemos usar o coringa “_”, como vemos na consulta 12. Nela, ‘5’ deve ser o terceiro caracter da string (de acordo com o nosso formato para data), de forma que usamos o valor ‘_5_____’, com cada sublinhado servindo como um substituto de um único caracter arbitrário dentro do campo procurado.

Consulta 12: encontre todos os empregados nascido durante a década de 1950.

```
SELECT
    prenome, sobrenome, dataNascimento
FROM
    Empresa.EMPREGADO
WHERE
    dataNascimento LIKE '_5_____';
```

	prenome	sobrenome	dataNascimento
1	Franklin	Wong	1955-12-08

Se um _ ou um % é necessário como um caracter literal na string, o caracter _ ou % deve ser precedido por um *caracter de escape*, que é especificado depois da string pelo uso da palavra reservada **ESCAPE**. Por exemplo, ‘AB\CD%\EF’ ESCAPE ‘\’ representa a string literal ‘AB_CD%\EF’ porque \ foi especificado como o caracter de escape.

Qualquer caracter não usado na string pode ser escolhido como o caracter de escape. Além disso, nós precisarmos também de uma regra para especificar apóstrofes (aspas simples - ‘ ’), se elas precisarem ser incluídas na string, pois elas são usadas como delimitadores de início e fim de strings. Se um apóstrofe (‘) é necessário dentro da string, ele será representado como dois apóstrofes consecutivos (”) de forma que eles não serão interpretados como o delimitador final da string:

```
SELECT
    prenome, sobrenome, dataNascimento
FROM
    Empresa.EMPREGADO
WHERE
    sobrenome like 'AB\_CD%\EF' ESCAPE '\';
```

[Vídeo adicional – Like e Not Like](#)

Você também pode usar operadores aritméticos na lista de campos solicitados no comando Select. Os operadores padrão são + (adição), - (subtração), * (multiplicação) e / (divisão) e podem ser aplicados a campos numéricos.

Por exemplo, suponha que queiramos ver o efeito de dar um aumento de 10% a cada empregado que trabalha no projeto 'ProjetoX'. A **consulta 13** mostrará o resultado disso para nós:

Consulta 13: mostre os salários atuais e os resultantes quando se aplica um aumento de 10% aos salários dos empregados que trabalham no projeto 'ProjetoX':

```
SELECT
    E.prenome, E.sobrenome,
    E.salario as SalarioAtual, 1.1 * E.salario AS SalarioAumentado
FROM
    Empresa.EMPREGADO AS E, Empresa.TRABALHA_EM AS T, Empresa.PROJETO AS P
WHERE
    E.NumSegSocial=T.NumSegSocial AND T.NumProjeto=P.NumProjeto AND
    P.NomeProjeto='ProdutoX';
```

O resultado vem abaixo:

	prenome	sobrenome	SalarioAtual	SalarioAumentado
1	John	Smith	30000.00	33000.000
2	Joyce	English	25000.00	27500.000

Podemos também aplicar o operador + a duas strings para concatená-las, ou seja, gerar uma nova string composta pelas duas originais. Abaixo fazemos isso na **Consulta 13b**, onde concatenamos prenome, um espaço e sobrenome para gerar o campo "Nome do Empregado":

Consulta 13b:

```
SELECT
    E.prenome+' '+E.sobrenome as 'Nome do Empregado',
    E.salario as SalarioAtual, 1.1 * E.salario AS SalarioAumentado
FROM
    Empresa.EMPREGADO AS E, Empresa.TRABALHA_EM AS T, Empresa.PROJETO AS P
WHERE
    E.NumSegSocial=T.NumSegSocial AND T.NumProjeto=P.NumProjeto AND
    P.NomeProjeto='ProdutoX';
```

O resultado dessa consulta vem abaixo:

	Nome do Empregado	SalarioAtual	SalarioAumentado
1	John Smith	30000.00	33000.000
2	Joyce English	25000.00	27500.000

Os operadores + e – podem também ser aplicados a campos de tipo Date, Datetime e Timestamp para incrementar (+) ou decrementar (-) intervalos de tempo em datas. Da mesma maneira, um intervalo de tempo é o resultado da diferença (-) entre duas datas.

Outro operador de comparação muito importante é o **Between**, ilustrado na **Consulta 14**:

Consulta 14: Recupere (retorne) número de segurança social, nome completo, sexo e salário dos empregados do departamento 5 cujos salários estejam entre 30000 e 40000:

```

SELECT
    NumSegSocial, prenome+' '+inicialMeio+' '+sobrenome Nome, Sexo, Salario, numDept
FROM
    Empresa.EMPREGADO
WHERE
    (Salario BETWEEN 30000 AND 40000) AND NumDept = 5;

```

	numSegSocial	Nome	Sexo	Salario	numDept
1	123456789	John B Smith	M	30000.00	5
2	333445555	Franklin T Wong	M	40000.00	5
3	666884444	Ramesh K Narayan	M	38000.00	5

[Vídeo adicional – operador Between](#)

4.6.6. Ordenação de Resultados de Consultas

SQL permite ao usuário do banco de dados ordenar os registros resultantes das consultas, pelo valor de um ou mais dos campos que aparecem no conjunto de resultados. Isso é feito pela cláusula ORDER BY e a ordenação pode ser feita em ordem crescente (o padrão) ou decrescente. A consulta 15 ilustra esse conceito:

Consulta 15: recupere uma lista de empregados e os projetos em que eles atuam, ordenada pelo departamento e, dentro de cada departamento, ordenado alfabeticamente pelo sobrenome em ordem decrescente e, depois, pelo prenome em ordem crescente.

```

SELECT D.NomeDept, E.Sobrenome, E.PreNome, P.NomeProjeto
FROM Empresa.DEPARTAMENTO D, Empresa.EMPREGADO E,
      Empresa.TRABALHA_EM T, Empresa.PROJETO P
WHERE
    D.NumDept= E.NumDept AND E.NumSegSocial= T.NumSegSocial AND
    T.NumProjeto= P.NumProjeto
ORDER BY
    D.NomeDept, E.Sobrenome DESC,
    E.PreNome;

```

Vemos o resultado ordenado ao lado.

A palavra reservada **DESC** indica que o campo será usado como chave de ordenação decrescente (descending, em inglês).

Pode-se usar a palavra **ASC** (ascending, em inglês) para indicar ordenação em ordem crescente, mas essa palavra é opcional.

Exemplo:

```

ORDER BY
    D.NomeDept ASC, E.Sobrenome DESC,
    E.PreNome;

```

[Vídeo adicional – Order By](#)

Suponha que você desejasse recuperar apenas os 10 primeiros registros de uma consulta qualquer. Isso poderia ser útil caso você deseje uma amostra de dados apenas, e não todos, ou tenha ordenado vários dados por salário e queira apenas os registros com os funcionários que tenham os dois maiores salários.

	NomeDept	Sobrenome	PreNome	NomeProjeto
1	Administração	Zelaya	Alicia	Informatização
2	Administração	Zelaya	Alicia	Novos Benefícios
3	Administração	Wallace	Jennifer	Reorganização
4	Administração	Wallace	Jennifer	Novos Benefícios
5	Administração	Jabbar	Ahmad	Informatização
6	Administração	Jabbar	Ahmad	Novos Benefícios
7	Pesquisa	Wong	Franklin	ProdutoY
8	Pesquisa	Wong	Franklin	ProdutoZ
9	Pesquisa	Wong	Franklin	Informatização
10	Pesquisa	Wong	Franklin	Reorganização
11	Pesquisa	Smith	John	ProdutoX
12	Pesquisa	Smith	John	ProdutoY
13	Pesquisa	Narayan	Ramesh	ProdutoZ
14	Pesquisa	English	Joyce	ProdutoX
15	Pesquisa	English	Joyce	ProdutoY
16	Sede	Borg	James	Reorganização
17	Tecn Informação	Camargo	Igor	Reorganização
18	Tecn Informação	Assis	Luis	Informatização

Para realizar essa filtragem, o comando SELECT tem a cláusula TOP, que vemos em funcionamento abaixo:

```
SELECT
    TOP 5 prenome, sobrenome, salario
from
    Empresa.EMPREGADO
order by
    Salario desc
```

		Resultados	Mensagens	
		prenome	sobrenome	salario
1		James	Borg	55000.00
2		Jennifer	Wallace	43000.00
3		Franklin	Wong	40000.00
4		Ramesh	Narayan	38000.00
5		John	Smith	30000.00

Há também a palavra **Percent** que, associada a Top, retorna os primeiros x por cento que forem indicados no Top. Abaixo recuperamos os 25% de empregados, ordenados em ordem decrescente dos maiores salários:

```
SELECT
    TOP 25 Percent prenome, sobrenome, salario
from
    Empresa.EMPREGADO
order by
    Salario desc
```

		Resultados	Mensagens	
		prenome	sobrenome	salario
1		James	Borg	55000.00
2		Jennifer	Wallace	43000.00
3		Franklin	Wong	40000.00

Como temos 11 empregados, 25% corresponde a 2,75 empregados, e o Sql Server arredonda esse valor para 3. Portanto, o result set acima, aplicado ao nosso conjunto de registros de empregados, retorna 3 registros.

Portanto, a cláusula **TOP** da linguagem SQL nos permite limitar o número de registros retornados por uma consulta e pode, por exemplo, garantir certo ganho de desempenho em algumas consultas que normalmente seriam compostas por uma quantidade muito grande de registros que deveriam trafegar pela rede caso todos eles fossem trazidos ao cliente.

Você pode usar TOP para limitar a quantidade de registros retornado por uma consulta interna, que é uma assunto que estudaremos mais à frente. Em certas consultas avançadas e complexas isso poderá ser útil.

Top pode ser usado também em comandos de manipulação de dados, como Insert, Update e Delete.

Para saber mais sobre o uso de Top no Sql Server, poderá consultar este [link](#).

[Vídeo adicional – cláusula Top – quantidade de registros retornados em um select](#)

Exercícios

Provenientes do livro Fundamental of Data Base Management Systems, de Ramez Elmasri e Shamkant B. Navathe

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figura 1 – banco de dados de exemplo de Escola

STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

GRADE_REPORT

Student_number	Section_identifier	Grade
----------------	--------------------	-------

Figura 2 – Tabelas e esquema simplificado do banco de dados de Escola

AIRPORT

Airport_code	Name	City	State
--------------	------	------	-------

FLIGHT

Flight_number	Airline	Weekdays
---------------	---------	----------

FLIGHT_LEG

Flight_number	Leg_number	Departure_airport_code	Scheduled_departure_time
		Arrival_airport_code	Scheduled_arrival_time

LEG_INSTANCE

Flight_number	Leg_number	Date	Number_of_available_seats	Airplane_id
		Departure_airport_code	Departure_time	Arrival_airport_code
		Arrival_time		

FARE

Flight_number	Fare_code	Amount	Restrictions
---------------	-----------	--------	--------------

AIRPLANE_TYPE

Airplane_type_name	Max_seats	Company
--------------------	-----------	---------

CAN_LAND

Airplane_type_name	Airport_code
--------------------	--------------

AIRPLANE

Airplane_id	Total_number_of_seats	Airplane_type
-------------	-----------------------	---------------

SEAT_RESERVATION

Flight_number	Leg_number	Date	Seat_number	Customer_name	Customer_phone
---------------	------------	------	-------------	---------------	----------------

Figura 3 – Banco de Dados AIRLINE

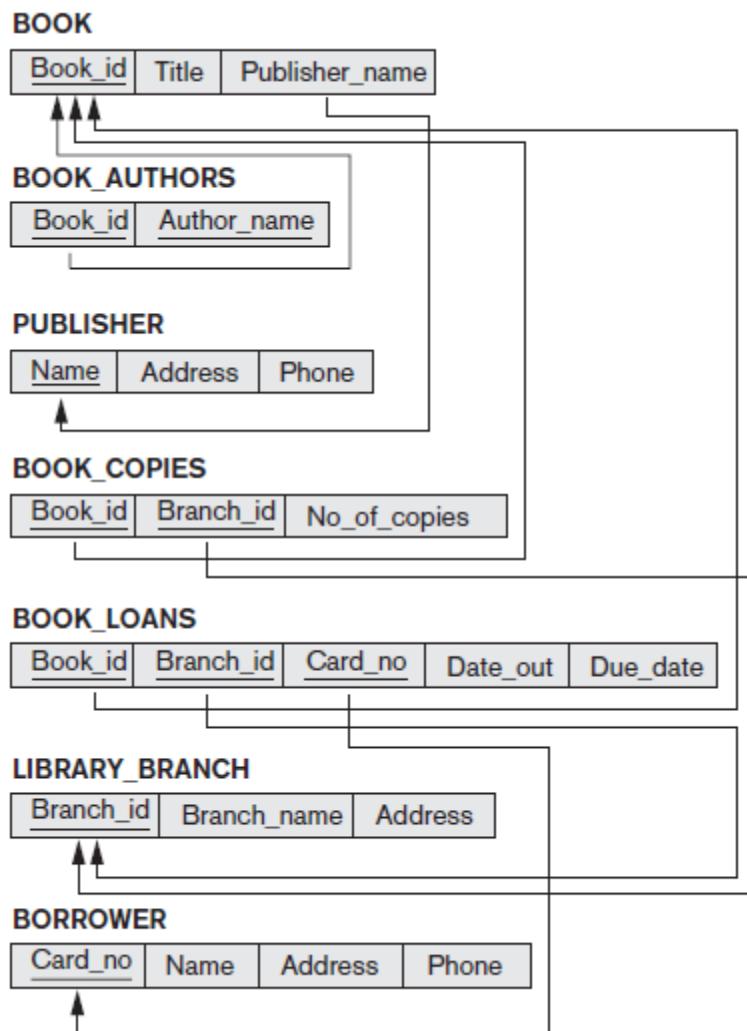


Figura 4 – Banco de Dados LIBRARY e seu esquema relacional

1. Considere o banco de dados mostrado na figura 1, cujo esquema simplificado é mostrado na figura 2.
 - a. Quais são as restrições de integridade referencial que deveriam ser mantidas no esquema?
 - b. Escreva comandos DDL de SQL apropriados para definir o banco de dados.
2. Repita o exercício 1, mas use o banco de dados AIRLINE da figura 3.
3. Considere o esquema relacional do banco de dados LIBRARY da figura 4.
 - a. Escolha a ação apropriada (restrict, cascade, set to NULL, set to default) para cada restrição de integridade referencial, tanto para a remoção de um registro referenciado quanto para a atualização do valor de um atributo chave primária em um registro referenciado por outro. Justifique suas escolhas.
 - b. Escreva comandos DDL de SQL apropriados para declarar o banco de dados LIBRARY segundo o esquema relacional da figura 4. Especifique as chaves e as ações referenciais definidas no item 3a.
4. Especifique as seguintes consultas SQL no banco de dados relacional EMPRESA, que usamos em nosso estudo e que está mostrado na [figura 4.7](#). Mostre o resultado de cada consulta se ela for aplicada ao banco de dados dessa figura.
 - a. Retorne os nomes de todos os empregados do departamento 5 que trabalham mais que 20 horas por semana no projeto ProdutoX.
 - b. Liste os nomes de todos os empregados que tenham um dependente com o mesmo prenome que o próprio empregado.

- c. Encontre os nomes de todos os empregados que são diretamente supervisionados por ‘Franklin Wong’.
5. Codifique os comandos SQL adequados para realizar as operações abaixo no banco de dados EMPRESA. Discuta todas as restrições de integridade violadas por cada operação, se houver, e as diferentes maneiras de impor essas restrições.
- Insira <‘Robert’, ‘F’, ‘Scott’, ‘943775543’, ‘1972-06-21’, ‘2365 Newcastle Rd, Bellaire, TX’, M, 58000, ‘888665555’, 1> em EMPREGADO
 - Insira <‘ProdutoA’, 4, ‘Bellaire’, 2> em PROJETO
 - Insira <‘Produção’, 4, ‘943775543’, ‘2007-10-01’> em DEPARTAMENTO.
 - Insira <‘677678989’, NULL, ‘40.0’> em TRABALHA_EM.
 - Insira <‘453453453’, ‘John’, ‘M’, ‘1990-12-12’, ‘esposo’> em DEPENDENTE.
 - Remova os registros de TRABALHA_EM com NumSegSocial = ‘333445555’.
 - Remova os registros de EMPREGADO com NumSegSocial = ‘987654321’.
 - Remova os registros de PROJETO com Nome de Projeto = ‘ProdutoX’.
 - Modifique os campos Gerente_numSegSocial e Gerente_dataInicial do registro de DEPARTAMENTO com NumDept = 5 para os valores ‘123456789’ e ‘2007-10-01’, respectivamente.
 - Modifique o campo Super_numSegSocial do registro de EMPREGADO com NumSegSocial = ‘999887777’ para o valor ‘943775543’.
 - Modifique o campo Horas de TRABALHA_EM com NumSegSocial = ‘999887777’ e NumProjeto = 10 para 5.0.

4.6.7. Consultas Avançadas

Existem vários recursos adicionais para consultas (SELECT) na linguagem SQL, que discutiremos a seguir.

Comparações envolvendo NULL e Lógica Tri-valorada

SQL tem várias regras para lidar com valores NULL. Lembre-se que NULL é usado para representar um valor não atribuído, mas que ele tem, geralmente, uma das três diferentes interpretações abaixo:

- **Valor desconhecido** (existe, mas não é conhecido): a data de nascimento de uma pessoa não é conhecida, então é representada por NULL no banco de dados.
- **Valor não disponível ou oculto** (existe, mas é ocultado propositalmente): uma pessoa tem um telefone residencial, mas não quer que ele seja listado, de forma que ele é mantido oculto e representado por NULL no banco de dados;
- **Valor não aplicável** (o atributo é indefinido para este registro): um atributo UltimoGrauUniversitario poderá ser NULL para uma pessoa que não tem graus em nível superior, porque esse atributo não se aplica a essa pessoa.

Frequentemente é impossível determinar qual dos significados acima é desejado; por exemplo, um NULL para o telefone residencial pode ter quaisquer dos três significados. Portanto, SQL não distingue entre esses diferentes significados de NULL.

Em geral, cada valor individual NULL é considerado diferente de cada um dos outros valores NULL nos diversos registros de um banco de dados.

Quando um NULL participa de uma operação de comparação, o resultado é considerado como UNKNOWN (DESCONHECIDO → ele pode ser VERDADERO ou ele pode ser FALSO). Por isso, SQL usa lógica tri-valorada com valores TRUE, FALSE e UNKNOWN ao invés dos dois valores padrão da lógica binária (Boolean) com os valores TRUE ou FALSE.

É, portanto, necessário definir os resultados (ou valores de verdade) de expressões lógicas tri-valoradas quando os conectivos lógicos AND, OR e NOT são usados com NULL. As tabelas abaixo mostram os valores resultantes:

(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

Nessas tabelas, as linhas e colunas representam os valores dos resultados de condições de comparação que, em geral, apareceriam em cláusulas WHERE de uma consulta SQL. Cada condição resultaria em um valor TRUE, FALSE ou UNKNOWN.

O resultado da combinação de dois valores (linha e coluna) pelo uso do conector lógico AND é mostrado na tabela (a) e o mesmo resultado pelo uso do conector lógico OR é mostrado em (b).

Por exemplo, TRUE AND UNKNOWN resulta em UNKNOWN, FALSE AND TRUE resulta em FALSE, UNKNOWN OR FALSE resulta em UNKNOWN.

Já a tabela (c) mostra que o uso de NOT inverte o resultado, a menos de UNKNOWN: NOT UNKNOWN continua sendo UNKNOWN.

Em consultas SQL, a regra geral é que somente aquelas combinações de registros que são avaliados, num WHERE, como TRUE é que serão selecionados para retorno ao usuário. Combinações de registros cujo WHERE resulte em FALSE ou UNKNOWN, não serão retornadas.

No entanto, existem possibilidades de obter exceções a essa regra, como no uso de OUTER JOIN, que estudaremos ainda.

SQL permite consultas que verificam se o valor de um atributo é NULL. Ao invés de usar = ou <> para comparar um valor de atributo com NULL, SQL usa o operador de comparação IS ou IS NOT. Isso ocorre porque SQL considera que cada valor NULL de um atributo é diferente de quaisquer outros valores NULL de quaisquer atributos, de forma que comparações de igualdade seriam inapropriadas. Portanto, disso segue que, quando uma condição de junção é especificada, registros com valores NULL nos campos da condição de junção não serão incluídos no result set (a menos que se use OUTER JOIN, como veremos logo mais).

Abaixo temos um exemplo de consulta nessa situação:

```
SELECT
    Prenome, Sobrenome
FROM
    Empresa.EMPREGADO
WHERE
    Super_numSegSocial IS NULL;
    --isNull(Super_numSegSocial,0) = 0
```

Resultados		Mensagens
	Prenome	Sobrenome
1	Cecilia	Kolonsky
2	James	Borg

Consultas Aninhadas (Subconsultas)

Algumas consultas exigem que os valores existentes em um banco de dados sejam recuperados e, então, usados em uma condição de comparação. Tais consultas podem ser convenientemente formuladas pelo uso de consultas aninhadas, que são blocos completos de comando Select-From-Where dentro da cláusula WHERE de uma outra consulta.

'Essa outra consulta é chamada de consulta externa. A [consulta 10](#) vista anteriormente foi formulada com o uso de UNION e sem uma consulta aninhada, mas ela pode ser refeita para usar consultas aninhadas como mostrado abaixo, na consulta 10a. Essa consulta 10a introduz o uso do operador de comparação IN, que compara um valor **c** com um conjunto de valores **V** e resulta em verdadeiro se **c** é um dos elementos de **V** (em outras palavras, **se c pertence a V**).

Consulta 10a: faça uma lista de todos os números de projeto que envolvem um empregado cujo sobrenome seja 'Smith', tanto como um funcionário ou como gerente do departamento que controla o projeto.

```
SELECT
    DISTINCT NumProjeto
FROM
    Empresa.PROJETO
WHERE
    NumProjeto IN      -- NumProjeto pertence ao conjunto abaixo?
    (
```

Resultados	
	Pnumber
1	1
2	2

```
    SELECT
        NumProjeto
    FROM
        Empresa.PROJETO P, Empresa.DEPARTAMENTO D, Empresa.EMPREGADO E
    WHERE
        P.NumDept= D.NumDept AND Gerente_numSegSocial=NumSegSocial
        AND Sobrenome='Smith'
```

```
) OR
NumProjeto IN      -- NumProjeto pertence ao conjunto abaixo?
(
```

```
    SELECT
        NumProjeto
    FROM
        Empresa.TRABALHA_EM T, Empresa.EMPREGADO E
    WHERE
        T.NumSegSocial = E.NumSegSocial AND Sobrenome='Smith'
```

```
);
```

A primeira consulta aninhada seleciona os números de projetos que tiveram um empregado com sobrenome 'Smith' envolvido como gerente, enquanto a segunda consulta aninhada seleciona os números de projetos que tem um empregado com sobrenome 'Smith' envolvido como empregado. Na consulta externa, usamos o operador lógico **OR** para recuperar um registro de PROJETO se o valor do NumProjeto desse registro pertence (está em) ao resultado de qualquer uma das consultas aninhadas.

Se uma consulta aninhada retorna um único atributo e um único registro, o resultado da consulta será um único valor escalar. Em tais casos, é permitido usar o operador = ao invés de IN para a comparação. Em geral, a consulta aninhada retornará uma tabela (relação de linhas) que é um conjunto (ou multiconjunto) de registros.

SQL, em algumas implementações, permite o uso de valores de registros em comparações pela colocação deles dentro de parênteses. Para ilustrar esse conceito, considere a consulta abaixo:

```
SELECT DISTINCT NumSegSocial
FROM
    Empresa.TRABALHA_EM
WHERE
    (NumProjeto, Horas) IN
```

```

(
    SELECT
        NumProjeto, Horas
    FROM
        Empresa.TRABALHA_EM
    WHERE
        NumSegSocial = '123456789'
);

```

No entanto, essa construção **não é aceita na versão mais atual do SQL Server**, embora funcione em Oracle, Postgre-SQL e MySQL. Para mais informações, clique [aqui](#).

Além do operador IN, outros operadores de comparação podem ser usados para comparar um valor v único (geralmente um campo de tabela) com um conjunto V (geralmente uma consulta aninhada). O operador = ANY (ou = SOME) retorna TRUE se o valor v é igual a algum valor do conjunto V e, portanto, equivale a IN. As palavras ANY e SOME tem o mesmo efeito. Outros operadores que podem ser combinados com ANY ou SOME incluem >, >=, <, <= e <>.

A palavra reservada ALL pode ser combinada com cada um desses operadores, também. Por exemplo, a condição de comparação ($v > \text{ALL } V$) retorna TRUE se o valor v é maior que todos os valores no conjunto V. Um exemplo é a consulta abaixo, que retorna os nomes dos empregados cujo salário é maior que o salário de todos os empregados do departamento 5:

```

SELECT
    Sobrenome, PreNome, Salario
FROM
    Empresa.EMPREGADO
WHERE
    Salario > ALL (
        SELECT
            Salario
        FROM
            Empresa.EMPREGADO
        WHERE
            NumDept=5
    );

```

	Sobrenome	PreNome	Salario
1	Borg	James	55000.00
2	Wallace	Jennifer	43000.00

Essa consulta também pode ser formulada usando a função de agregação MAX, que estudaremos mais à frente.

Em geral, podemos ter vários níveis de consultas aninhadas. Podemos novamente enfrentar a possibilidade de ambiguidade entre nomes de atributos se atributos com o mesmo nome existirem – um na tabela da cláusula FROM da consulta externa, e outra na tabela da cláusula FROM da consulta aninhada (interna). A regra estabelece que uma referência a um atributo não-qualificado se refere à tabela declarada na consulta mais interna de todas. Por exemplo, na cláusula SELECT e na cláusula WHERE da primeira consulta interna da Consulta 10a, a referência a qualquer atributo não qualificado da tabela PROJETO se refere à relação PROJETO especificada na cláusula FROM da consulta aninhada (**interna**).

Para se referir a um atributo da tabela PROJETO da consulta externa, devemos especificar e referenciar um alias para a relação. Essas regras são similares a regras de escopo para variáveis de programa na maioria das linguagens de programação que permitem métodos (procedimentos e funções) aninhados. Para ilustrar a potencial ambiguidade de nomes de atributos em consultas aninhadas, veja a consulta abaixo:

Consulta 16: Recupere o nome de cada empregado que tenha um dependente com o mesmo prenome e o mesmo sexo que o empregado.

```

SELECT E.PreNome, E.Sobrenome
FROM

```

```

    Empresa.EMPREGADO AS E
WHERE
    E.NumSegSocial IN (
        SELECT
            NumSegSocial
        FROM
            Empresa.DEPENDENTE AS D
        WHERE
            E.PreNome=D.NomeDependente AND E.Sexo=D.Sexo
);

```

	PreNome	Sobrenome
1	John	Smith
2	Franklin	Wong

Na consulta aninhada acima, devemos qualificar E.Sexo porque ele se refere ao atributo Sexo de EMPREGADO da consulta externa, e DEPENDENTE também tem um atributo com o mesmo nome (Sexo). Se houvesse qualquer referência não qualificada a Sexo na consulta interna, ela se referiria ao atributo Sexo de DEPENDENTE.

No entanto, não teríamos de qualificar os atributos prenome e numSegSocial de EMPREGADO, se eles aparecessem na consulta interna porque a tabela DEPENDENTE não tem atributos com esses nomes, de forma que não haveria ambiguidade.

Em geral, é aconselhável criar **alias** (apelidos) para todas as tabelas referenciadas em uma consulta SQL para evitar potenciais erros e ambiguidades, como fizemos na consulta 16, acima.

Consultas Aninhadas Correlacionadas

Sempre que uma condição na cláusula WHERE de uma consulta aninhada referencia algum atributo de uma tabela declarada na consulta externa, as duas consultas são ditas como sendo correlacionadas.

Podemos entender melhor esse conceito considerando que a consulta interna (aninhada) é avaliada uma vez para cada registro (ou combinação de registros) da consulta externa. Por exemplo, podemos pensar da consulta 16 como se segue:

Para cada registro de EMPREGADO, avalie a consulta aninhada, que recupera os valores de NumSegSocial para todos os registros de DEPENDENTE com o mesmo sexo e prenome que o registro de EMPREGADO atual. Se o valor NumSegSocial do registro do EMPREGADO está no conjunto resultante da consulta interna, então selecione esse registro de EMPREGADO.

Em geral, uma consulta escrita com um bloco select-from-where aninhado e usado os operadores de comparação = ou IN, sempre podem ser expressos como uma consulta de um único bloco. Por exemplo, a consulta 16 pode ser reescrita como:

Consulta 16a: Recupere o nome de cada empregado que tenha um dependente com o mesmo prenome e o mesmo sexo que o empregado.

```

SELECT
    E.PreNome, E.Sobrenome
FROM
    Empresa.EMPREGADO AS E,
    Empresa.DEPENDENTE AS D
WHERE
    E.NumSegSocial=D.NumSegSocial AND E.Sexo=D.Sexo AND
    E.PreNome=D.NomeDependente;

```

	PreNome	Sobrenome
1	John	Smith
2	Franklin	Wong

[Vídeo adicional – Operadores In e Not In](#)

[Vídeo adicional - Subconsultas](#)

[Funções EXISTS e UNIQUE do SQL](#)

A função **EXISTS** em SQL é usada para verificar se o resultado de uma consulta aninhada correlacionada é vazio (não contém nenhum registro) ou não. **EXISTS** tem como resultado um valor lógico: TRUE se o resultado da consulta aninhada contém ao menos um registro, ou FALSE se esse resultado não tem nenhum registro. Vamos usar a consulta 16 novamente como modelo, mas reformulada como abaixo:

Consulta 16b: Recupere o nome de cada empregado que tenha um dependente com o mesmo prenome e o mesmo sexo que o empregado.

```
SELECT
    E.PreNome, E.Sobrenome
FROM
    Empresa.EMPREGADO AS E
WHERE
    EXISTS (
        SELECT
            *
        FROM
            Empresa.DEPENDENTE AS D
        WHERE
            E.NumSegSocial=D.NumSegSocial AND E.Sexo=D.Sexo AND
            E.PreNome=D.NomeDependente
    );
```

Resultados		
	PreNome	Sobrenome
1	John	Smith
2	Franklin	Wong

EXISTS e NOT EXISTS são tipicamente usados em conjunto com uma consulta aninhada correlacionada. Na consulta 16b, a consulta aninhada referencia os atributos NumSegSocial, PreNome e Sexo da tabela EMPREGADO pertencentes à consulta externa.

Podemos imaginar a Consulta 16b como segue: para cada registro de EMPREGADO, avalie a consulta aninhada, que recupera todos os registros de DEPENDENTE com o mesmo NumSegSocial, Sexo e NomeDependente que o registro de EMPREGADO; se ao menos um registro EXISTE no resultado da consulta aninhada, então selecione aquele registro de EMPREGADO para compor o resultado retornado pelo SELECT.

Em geral, EXISTS(Q) retorna **TRUE** se há ao menos um registro no resultado da consulta aninhada Q, e retorna **FALSE** caso contrário. Por outro lado, NOT EXISTS(Q) retorna **TRUE** se não há nenhum registro no resultado da consulta aninhada Q, e retorna **FALSE** caso contrário. Abaixo, ilustramos o uso de NOT EXISTS:

Consulta 17: Recupera os nomes dos empregados que não tenham dependentes.

```
SELECT
    PreNome, Sobrenome
FROM
    Empresa.EMPREGADO E
WHERE
    NOT EXISTS (
        SELECT *
        FROM Empresa.DEPENDENTE D
        WHERE E.NumSegSocial = D.NumSegSocial
    );
```

Resultados		
	PreNome	Sobrenome
1	Joyce	English
2	Luis	Assis
3	Ramesh	Narayan
4	Cecilia	Kolonsky
5	James	Borg
6	Ahmad	Jabbar
7	Alicia	Zelaya

Na Consulta 17, a consulta aninhada correlacionada recupera todos os registros de DEPENDENTE relacionados a um EMPREGADO específico. Se **não existe** nenhum registro de dependente resultante dessa busca, o registro do EMPREGADO é selecionado porque a cláusula WHERE será

avaliada como TRUE, nesse caso (NOT False = True). Podemos explicar a **Consulta 17** como se segue:

Para cada registro de EMPREGADO, a consulta aninhada correlacionada seleciona todos os registros de DEPENDENTE cujo NumSegSocial combina com o NumSegSocial do EMPREGADO; se o resultado for vazio, nenhum dependente é relacionado com esse empregado, de forma que esse empregado é selecionado e seus campos PreNome e Sobrenome são recuperados e retornados no conjunto de resultados da consulta.

Temos abaixo outra consulta com a função EXISTS:

Consulta 18: Liste os nomes dos gerentes que tenham ao menos um dependente.

```
SELECT
    PreNome, Sobrenome
FROM
    Empresa.EMPREGADO Emp
WHERE EXISTS (
    SELECT *
    FROM Empresa.DEPENDENTE Dep
    WHERE Emp.NumSegSocial=Dep.NumSegSocial
)
AND EXISTS (
    SELECT *
    FROM Empresa.DEPARTAMENTO Depto
    WHERE
        Emp.NumSegSocial=Dept.Gerente_numSegSocial
);
;
```

	PreNome	Sobrenome
1	Franklin	Wong
2	Jennifer	Wallace

Conjuntos Explícitos e Renomeação de Atributos em SQL

Já usamos diversas consultas com consultas aninhadas na cláusula WHERE. Além disso, na cláusula WHERE também é possível usar um conjunto explícito de valores, ou seja, sem que os valores (elementos) desse conjunto sejam resultantes de uma consulta aninhada. Esse conjunto é uma lista de valores, separados por vírgulas e delimitados por parênteses.

Consulta 19: Recupere o Número de Seguridade Social (NumSegSocial) de todos os empregados que trabalham nos projetos de número 1, 2 ou 3.

```
SELECT
    DISTINCT NumSegSocial
FROM
    Empresa.TRABALHA_EM
WHERE
    NumProjeto IN (1, 2, 3);
```

	NumSegSocial
1	123456789
2	333445555
3	453453453
4	666884444

Já vimos em algumas outras consultas que podemos renomear os atributos e mesmo compor atributos com concatenação, dando um nome a esses atributos calculados. Isso é feito com o qualificador AS seguido pelo novo nome desejado. Portanto, AS pode ser usado para definir um apelido (ALIAS) tanto para tabelas (FROM) quanto para atributos (SELECT).

Por exemplo, a consulta 3a, abaixo, é uma reescrita da [Consulta 3](#), onde se recupera os nomes completos do empregado e de seu supervisor imediato, mas renomeando esses atributos. Os novos nomes aparecerão como os cabeçalhos de coluna do resultado da consulta:

Consulta 3a: para cada empregado, recuperar o prenome e sobrenome do empregado e o prenome e sobrenome do seu supervisor imediato.

```
SELECT E.PreNome+' '+E.Sobrenome AS Empregado,
       S.PreNome+' '+S.Sobrenome AS Supervisor
  FROM Empresa.EMPREGADO AS E,
       Empresa.EMPREGADO AS S
 WHERE E.Super_numSegSocial = S.NumSegSocial;
```

Abaixo temos o resultado da [Consulta 3](#) original:

	Prenome	Sobrenome	NomeSup	SobrenomeSup
1	John	Smith	Franklin	Wong
2	Franklin	Wong	James	Borg
3	Joyce	English	Franklin	Wong
4	Igor	Camargo	James	Borg
5	Luis	Assis	James	Borg
6	Ramesh	Narayan	Franklin	Wong
7	Jennifer	Wallace	James	Borg
8	Ahmad	Jabbar	Jennifer	Wallace
9	Alicia	Zelaya	Jennifer	Wallace

	Empregado	Supervisor
1	John Smith	Franklin Wong
2	Franklin Wong	James Borg
3	Joyce English	Franklin Wong
4	Igor Camargo	James Borg
5	Luis Assis	James Borg
6	Ramesh Narayan	Franklin Wong
7	Jennifer Wallace	James Borg
8	Ahmad Jabbar	Jennifer Wallace
9	Alicia Zelaya	Jennifer Wallace

4.6.8. Junção de Tabelas e Junções Externas (Outer Join)

O conceito de uma tabela unida (ou relação unida) ou tabela em junção foi incorporada ao SQL para permitir aos usuários especificar um conjunto de dados resultante diretamente na cláusula FROM de uma consulta, ao invés de definir a junção na cláusula WHERE.

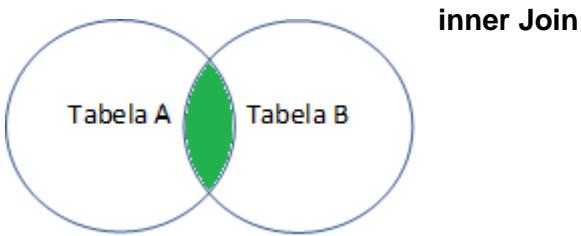
Essa forma de codificar a consulta facilita a sua compreensão, ao invés de misturar todas as condições de junção e de seleção na cláusula WHERE.

Por exemplo, considere a [consulta 1](#), que recupera o nome e endereço de todos os empregados que trabalham para o departamento ‘Pesquisa’. Originalmente, essa consulta misturava as condições de seleção e de junção na cláusula WHERE, como se pode ver [aqui](#). Essa é uma consulta simples, onde é relativamente fácil discernir a operação de seleção em relação à operação de junção mas, mesmo assim, é interessante separar funcionalmente as duas operações, pois isso facilitaria manutenções futuras do código da consulta e, também, evitaria que um erro numa das condições atrapalhasse o funcionamento da outra condição.

Abaixo temos a Consulta 1a, que reformula a [consulta 1](#), levando em consideração a ideia de separar a operação de junção da operação de seleção:

Consulta 1a: recupera o nome e o endereço de todos os empregados que trabalham no departamento ‘Pesquisa’:

```
SELECT Prenome, Sobrenome, Endereco
  FROM (
    Empresa.EMPREGADO E JOIN Empresa.DEPARTAMENTO D
      ON E.numDept = D.numDept
  )
 WHERE NomeDept = 'Pesquisa';
```



Pode-se ver o resultado da cláusula FROM como uma única tabela unida, ou seja, resultante da junção de duas tabelas através da **expressão de junção E.NumDept = D.NumDept**. O servidor de banco de dados junta as duas tabelas, combinando os registros das duas tabelas que possuem o mesmo número de departamento. Ou seja, realiza uma **interseção** entre as duas tabelas através da condição dada.

A tabela unida é uma tabela **virtual**, não física, que **existe apenas durante o processamento da consulta**. Os atributos da tabela unida são todos os atributos da primeira tabela, EMPREGADO, seguidos por todos os atributos da segunda tabela, DEPARTAMENTO.

Já a cláusula WHERE fica responsável por **selecionar**, dentro dessa tabela unida, apenas os registros cujo campo NomeDept seja igual a 'Pesquisa'.

A partir dos resultados do WHERE e do JOIN, a cláusula SELECT projeta (apresenta no conjunto de dados resultante) apenas os campos Prenome, Sobrenome e Endereco.

Na linguagem SQL padrão, o conceito de tabelas unidas permite também ao usuário especificar diferentes tipos de junção, tais como NATURAL JOIN e vários tipos de OUTER JOIN.

Em um **NATURAL JOIN** sobre duas tabelas R e S, nenhuma condição de junção é especificada: é criada uma condição implícita **EQUIJOIN** para cada par de atributos com o **mesmo nome** nas duas tabelas R e S. Cada par desses atributos é incluída somente uma vez no conjunto de dados resultante. Ou seja, em NATURAL JOIN não usamos a condição de junção especificada em ON.

Essa abordagem privilegia **esquemas de bancos de dados bem organizados**, onde **nomes de atributos iguais em diferentes tabelas possuem o mesmo significado**, ou seja, **armazenam informações da mesma natureza**. É mais ou menos o que acontece no banco de dados que estamos usando que, por motivos pedagógicos, teve campos com nomes diferentes tendo o mesmo significado (Gerente_numSegSocial e numSegSocial, por exemplo).

No entanto, no Sql Server, Natural Join **não é previsto**, de forma que você terá sempre de especificar a condição de junção, mesmo que os campos de junção das duas tabelas envolvidas tenham o mesmo nome (e é bom que tenham o mesmo nome, pois **significam a mesma informação**, dividida em duas tabelas diferentes durante a normalização). Para diferenciar um campo do outro, ou seja, saber de qual tabela se originam, usa-se o apelido da tabela. Nas duas tabelas acima o número do Departamento da tabela de Departamento e o número do departamento em que o empregado trabalha foram identificados pelo mesmo nome, numDept. Por esse motivo usamos os alias para identificar cada tabela ao qualificar campos de mesmo nome:

```

SELECT
    Prenome, Sobrenome, Endereco
FROM
    (
        Empresa.EMPREGADO as E JOIN Empresa.DEPARTAMENTO D -- AS optional
        ON E.numDept=D.numDept
    )
WHERE
    NomeDept='Pesquisa';

```

Apelidos das tabelas

O tipo default (padrão) de uma junção é chamado de **Inner Join**, ou junção interna, onde um registro é incluído no conjunto de resultados somente se houver um registro correspondente na outra tabela. Pode-se tanto escrever **Inner Join** quanto apenas **Join**, o resultado será o mesmo.

Por exemplo, a [consulta da página 94](#) pode ser reescrita como abaixo, usando Inner Join. O resultado dessa consulta inclui somente empregados que tenham um supervisor. Empregados sem supervisores não são incluídos no conjunto de dados resultante:

Consulta 20: Relacione todos os nomes e sobrenomes dos empregados supervisionados por algum outro empregado, além de relacionar os nomes e sobrenomes dos supervisores:

```
SELECT E.NumSegSocial 'E.NumSegSocial', E.Prenome Prenome, E.Sobrenome,
       E.Super_numSegSocial as 'E.Super_numSegSocial',
       S.NumSegSocial 'S.NumSegSocial', S.PreNome NomeSup,
       S.Sobrenome SobrenomeSup
  FROM (
    Empresa.EMPREGADO AS E INNER JOIN Empresa.EMPREGADO AS S
    ON E.Super_numSegSocial = S.NumSegSocial
);
```

	E.NumSegSocial	Prenome	Sobrenome	E.Super_numSegSocial	S.NumSegSocial	NomeSup	SobrenomeSup
1	123456789	John	Smith	333445555	333445555	Franklin	Wong
2	333445555	Franklin	Wong	888665555	888665555	James	Borg
3	453453453	Joyce	English	333445555	333445555	Franklin	Wong
4	494549454	Igor	Camargo	888665555	888665555	James	Borg
5	512851285	Luis	Assis	888665555	888665555	James	Borg
6	666884444	Ramesh	Narayan	333445555	333445555	Franklin	Wong
7	987654321	Jennifer	Wallace	888665555	888665555	James	Borg
8	987987987	Ahmad	Jabbar	987654321	987654321	Jennifer	Wallace
9	999887777	Alicia	Zelaya	987654321	987654321	Jennifer	Wallace

Na figura acima, resultante da consulta acima, somente os registros de empregados que possuem um supervisor foram selecionados, pois a junção interna (Inner Join ou somente Join) leva em consideração a comparação de igualdade entre os campos Super_NumSegSocial e NumSegSocial. Se o Super_NumSegSocial é null, ou seja, esse empregado não tem nenhum supervisor, esse registro não terá seu registro no resultado da consulta e não será exibido.

Em outras palavras, temos acima apenas os empregados supervisionados por outro empregado.

[Vídeo adicional – Inner Join](#)

Se quiséssemos que todos os funcionários, mesmo aqueles sem supervisor, fossem selecionados, um Inner Join não funcionaria. Para esse tipo de resultado, onde os relacionamentos “quebrados” são também listados, é necessário o uso de uma **Junção Externa**, ou **Outer Join**.

As tabelas em um **Outer Join** são classificadas como Esquerda (**LEFT**) e Direita (**RIGHT**), conforme a ordem de aparecimento na operação de junção. Se você deseja que o registro da tabela da **esquerda** seja selecionado, mesmo que não haja um registro correspondente a ele na tabela da direita, você especifica a junção como **LEFT OUTER JOIN**.

Vemos isso abaixo, onde os registros de EMPREGADO aparecerão, mesmo que não tenham um supervisor, junto com os registros de EMPREGADO que tenham um supervisor:

Consulta 21: Relacione todos os nomes e sobrenomes dos empregados e de seus respectivos supervisores. Se um empregado não tiver um supervisor, liste apenas os dados do empregado, deixando os do supervisor não listados (já que esse empregado existe, mas não tem supervisor):

SELECT

```
E.NumSegSocial as 'E.NumSegSocial', E.Prenome, E.Sobrenome,
E.Super_numSegSocial as 'E.Super_numSegSocial',
S.NumSegSocial as 'S.NumSegSocial', S.Prenome as NomeSup,
S.Sobrenome as SobreNomeSup
FROM
(
    Empresa.EMPREGADO as E LEFT OUTER JOIN Empresa.EMPREGADO as S
    ON E.Super_numSegSocial = S.NumSegSocial
);
```

	E.NumSegSocial	Prenome	Sobrenome	E.Super_numSegSocial	S.NumSegSocial	NomeSup	SobreNomeSup
1	123456789	John	Smith	333445555	333445555	Franklin	Wong
2	333445555	Franklin	Wong	888665555	888665555	James	Borg
3	453453453	Joyce	English	333445555	333445555	Franklin	Wong
4	494549454	Igor	Camargo	888665555	888665555	James	Borg
5	512851285	Luis	Assis	888665555	888665555	James	Borg
6	666884444	Ramesh	Narayan	333445555	333445555	Franklin	Wong
7	677678989	Cecilia	Kolonsky	NULL	NULL	NULL	NULL
8	888665555	James	Borg	NULL	NULL	NULL	NULL
9	987654321	Jennifer	Wallace	888665555	888665555	James	Borg
10	987987987	Ahmad	Jabbar	987654321	987654321	Jennifer	Wallace
11	999887777	Alicia	Zelaya	987654321	987654321	Jennifer	Wallace

Como os registros dos Empregados Cecilia e James não tem valor no campo Super_NúmeroSegSocial, não há correspondência entre a tabela E e a tabela S nesses registros. Nesse caso, são selecionados os registros da tabela à esquerda (LEFT) nessa junção

Na figura acima, os registros 7 e 8 se referem a empregados que não possuem um supervisor. Dessa forma, os dados da tabela à esquerda (LEFT) foram listados, mesmo que não tenham correspondentes na tabela da direita.

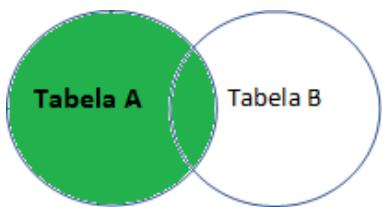
O mesmo resultado é obtido se invertermos as posições das tabelas na operação de junção e usarmos RIGHT, ao invés de LEFT:

Consulta 21a:

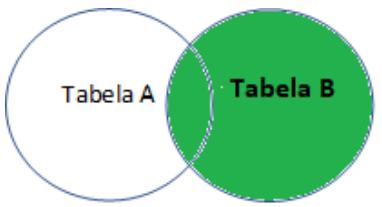
```
SELECT E.NumSegSocial as 'E.NumSegSocial', E.PreNome, E.Sobrenome,
E.Super_numSegSocial as 'E.Super_numSegSocial',
S.NumSegSocial as 'S.NumSegSocial', S.PreNome NomeSup,
S.Sobrenome SobreNomeSup
FROM (
    Empresa.EMPREGADO AS S RIGHT OUTER JOIN Empresa.EMPREGADO AS E
    ON E.Super_numSegSocial = S.NumSegSocial );
```

Observe, no resultado dessa consulta apresentado na figura abaixo, que a disposição dos campos no conjunto de dados resultante é definida na cláusula SELECT, e não reflete a disposição das tabelas na operação de junção:

	E.NumSegSocial	PreNome	Sobrenome	E.Super_numSegSocial	S.NumSegSocial	NomeSup	SobreNomeSup
1	123456789	John	Smith	333445555	333445555	Franklin	Wong
2	333445555	Franklin	Wong	888665555	888665555	James	Borg
3	453453453	Joyce	English	333445555	333445555	Franklin	Wong
4	494549454	Igor	Camargo	888665555	888665555	James	Borg
5	512851285	Luis	Assis	888665555	888665555	James	Borg
6	666884444	Ramesh	Narayan	333445555	333445555	Franklin	Wong
7	677678989	Cecilia	Kolonsky	NULL	NULL	NULL	NULL
8	888665555	James	Borg	NULL	NULL	NULL	NULL
9	987654321	Jennifer	Wallace	888665555	888665555	James	Borg
10	987987987	Ahmad	Jabbar	987654321	987654321	Jennifer	Wallace
11	999887777	Alicia	Zelaya	987654321	987654321	Jennifer	Wallace



Num **LEFT OUTER JOIN**, os registros referentes à tabela colocada à **esquerda** da operação de junção sempre aparecerão no resultado, mesmo que não tenham correspondentes na tabela da direita, de acordo com a expressão de junção. Os registros da tabela da direita cujos valores da expressão de junção não coincidirem com os valores dos mesmos atributos de nenhum registro da tabela da esquerda não serão combinados nesse tipo de junção.



Da mesma maneira, num **RIGHT OUTER JOIN**, os registros referentes à tabela colocada à **direita** da operação de junção sempre aparecerão no resultado, mesmo que não tenham correspondentes na tabela da esquerda, de acordo com a expressão de junção. Os registros da tabela da esquerda cujos valores da expressão de junção não coincidirem com os valores dos mesmos atributos de nenhum registro da tabela da direita não serão combinados nesse tipo de junção.

Se modificarmos o comando como abaixo, teremos um resultado bastante diferente:

Consulta 22:

```

SELECT
    E.NumSegSocial AS 'E.NumSegSocial', E.PreNome, E.Sobrenome,
    E.Super_numSegSocial AS 'E.Super_numSegSocial',
    S.NumSegSocial AS 'S.NumSegSocial', S.PreNome NomeSup,
    S.Sobrenome SobreNomeSup
FROM (
    Empresa.EMPREGADO AS E RIGHT OUTER JOIN Empresa.EMPREGADO AS S
    ON E.Super_numSegSocial = S.NumSegSocial
);

```

No resultado a seguir, todos os registros da tabela da direita foram listados, mesmo que não tenham nenhum empregado supervisionado. Alguns empregados apareceram mais de uma vez, porque possuem um ou mais empregados supervisionados por eles.

	E.NumSegSocial	Prenome	Sobrenome	E.Super_numSegSocial	S.NumSegSocial	NomeSup	SobreNomeSup
1	NULL	NULL	NULL	NULL	123456789	John	Smith
2	123456789	John	Smith	333445555	333445555	Franklin	Wong
3	453453453	Joyce	English	333445555	333445555	Franklin	Wong
4	666884444	Ramesh	Narayan	333445555	333445555	Franklin	Wong
5	NULL	NULL	NULL	NULL	453453453	Joyce	English
6	NULL	NULL	NULL	NULL	494549454	Igor	Camargo
7	NULL	NULL	NULL	NULL	512851285	Luis	Assis
8	NULL	NULL	NULL	NULL	666884444	Ramesh	Narayan
9	NULL	NULL	NULL	NULL	677678989	Cecilia	Kolonsky
10	333445555	Franklin	Wong	888665555	888665555	James	Borg
11	494549454	Igor	Camargo	888665555	888665555	James	Borg
12	512851285	Luis	Assis	888665555	888665555	James	Borg
13	987654321	Jennifer	Wallace	888665555	888665555	James	Borg
14	987987987	Ahmad	Jabbar	987654321	987654321	Jennifer	Wallace
15	999887777	Alicia	Zelaya	987654321	987654321	Jennifer	Wallace
16	NULL	NULL	NULL	NULL	987987987	Ahmad	Jabbar
17	NULL	NULL	NULL	NULL	999887777	Alicia	Zelaya

[Vídeo adicional – Outer Join](#)

Há, também, o **FULL OUTER JOIN**, em que os dois conjuntos acima são exibidos:

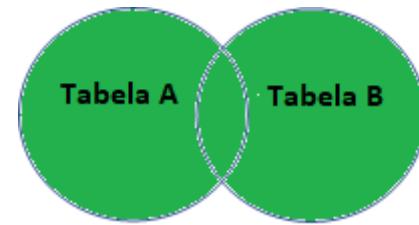
Consulta 23:

SELECT

```

E.NumSegSocial as 'E.NumSegSocial', E.Prenome,
E.Sobrenome,
E.Super_numSegSocial as 'E.Super_numSegSocial',
S.NumSegSocial as 'S.NumSegSocial', S.PreNome
NomeSup,
S.Sobrenome SobreNomeSup
FROM
(
    Empresa.EMPREGADO AS E FULL OUTER JOIN Empresa.EMPREGADO AS S
    ON E.Super_numSegSocial = S.NumSegSocial
);

```



	E.NumSegSocial	Prenome	Sobrenome	E.Super_numSegSocial	S.NumSegSocial	NomeSup	SobreNomeSup
1	123456789	John	Smith	333445555	333445555	Franklin	Wong
2	333445555	Franklin	Wong	888665555	888665555	James	Borg
3	453453453	Joyce	English	333445555	333445555	Franklin	Wong
4	494549454	Igor	Camargo	888665555	888665555	James	Borg
5	512851285	Luis	Assis	888665555	888665555	James	Borg
6	666884444	Ramesh	Narayan	333445555	333445555	Franklin	Wong
7	677678989	Cecilia	Kolonsky	NULL	NULL	NULL	NULL
8	888665555	James	Borg	NULL	NULL	NULL	NULL
9	987654321	Jennifer	Wallace	888665555	888665555	James	Borg
10	987987987	Ahmad	Jabbar	987654321	987654321	Jennifer	Wallace
11	999887777	Alicia	Zelaya	987654321	987654321	Jennifer	Wallace
12	NULL	NULL	NULL	NULL	123456789	John	Smith
13	NULL	NULL	NULL	NULL	453453453	Joyce	English
14	NULL	NULL	NULL	NULL	494549454	Igor	Camargo
15	NULL	NULL	NULL	NULL	512851285	Luis	Assis
16	NULL	NULL	NULL	NULL	666884444	Ramesh	Narayan
17	NULL	NULL	NULL	NULL	677678989	Cecilia	Kolonsky
18	NULL	NULL	NULL	NULL	987987987	Ahmad	Jabbar
19	NULL	NULL	NULL	NULL	999887777	Alicia	Zelaya

Como vemos, existe uma variedade de operações de junção externa na teoria relacional. Em SQL, as opções disponíveis para especificar tabelas unidas incluem

- **INNER JOIN** (o mesmo que JOIN): somente pares de registros que combinam entre si através da condição de junção são recuperados;
- **LEFT OUTER JOIN** (o mesmo que LEFT JOIN): cada registro na tabela da esquerda deve aparecer no resultado; se um registro não tem correspondente na tabela da direita, os campos da tabela da direita para esse registro de resultado são preenchidos com NULL;
- **RIGHT OUTER JOIN** (o mesmo que RIGHT JOIN): cada registro na tabela da direita deve aparecer no resultado; se um registro não tem correspondente na tabela da esquerda, os campos da tabela da direita para esse registro de resultado são preenchidos com NULL;
- **FULL OUTER JOIN** (o mesmo que FULL JOIN): todos os registros das tabelas são apresentados no resultado, tendo ou não correspondência; quando há correspondência, o registro resultante é composto pelos valores dos campos selecionados das duas tabelas; quando não há correspondência, campos de cada tabela são apresentados em registros diferentes no resultado, sendo os demais campos da outra tabela preenchidos com Null;
- **CROSS JOIN:** [produto cartesiano](#) das duas tabelas envolvidas, já estudado anteriormente. Esse tipo de junção não possui condição de junção, já que todos os registros das duas tabelas serão combinados entre si. No caso de nossas tabelas de exemplo, haveria mais de 120 registros no resultado. Esse tipo de operação deve ser evitado na grande maioria das vezes, pois a sua aplicação útil em situações normais de trabalho com os dados é bastante reduzida. A consulta abaixo demonstra essa operação. Note que não há condição de junção (cláusula ON)

Consulta 24:

```
SELECT
    E.NumSegSocial as 'E.NumSegSocial', E.Prenome as Prenome, E.Sobrenome,
    E.Super_numSegSocial as 'E.Super_numSegSocial',
    S.NumSegSocial as 'S.NumSegSocial', S.PreNome NomeSup,
    S.Sobrenome SobreNomeSup
FROM
    (
        Empresa.EMPREGADO AS E CROSS JOIN Empresa.EMPREGADO AS S
    );
```

[Artigo sobre JOIN](#)

[Fazendo Outer Join em Oracle com o operador +](#)

[Vídeo adicional – Full Join](#)

Junções Aninhadas

Em muitas situações, poderá ser necessário efetuar a junção de mais de duas tabelas. Nessas situações uma ou mais tabelas da junção será o resultado de outras junções. Assim, poderemos juntar três ou mais tabelas como uma única tabela unida, o que é chamado de junção múltipla.

Observe o comando de uma consulta semelhante à 2, vista anteriormente:

```
SELECT
    NumProjeto, P.NumDept, Sobrenome, Endereco, DataNascimento
FROM
    Empresa.PROJETO P, Empresa.DEPARTAMENTO D, Empresa.EMPREGADO E
WHERE
    P.NumDept = D.NumDept AND Gerente_numSegSocial = NumSegSocial AND
    LocalProjeto='Stafford';
```

Elas foram escritas sem o uso de JOIN, de forma que as condições de seleção e de junção estão misturadas na cláusula WHERE. Também envolve a junção de três tabelas.

Podemos reescrevê-las, usando JOIN para unir as três tabelas, duas a duas, como fazemos na consulta 2a, em seguida:

Consulta 2a: Para cada projeto localizado em ‘Stafford’, lista o número do projeto, o número do departamento que controla esse projeto e o sobrenome, endereço e data de nascimento do gerente do departamento:

```
SELECT NumProjeto, NumDept, Sobrenome, Endereco, DataNascimento
FROM (
    (
        Empresa.PROJETO P JOIN Empresa.DEPARTAMENTO D
        ON P.numDept = D.numDept
    )
    JOIN Empresa.EMPREGADO E
    ON Gerente_numSegSocial = NumSegSocial
)
WHERE LocalProjeto='Stafford';
```

	NumProjeto	NumDept	Sobrenome	Endereco	DataNascimento
1	10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
2	30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

O uso de parênteses nas operações de junção é opcional. No entanto, seu uso acaba auxiliando a melhorar a compreensão dos comandos e agrupar as diversas operações aninhadas de junção.

[Mais sobre Junção Múltipla no SQL Server](#)

4.6.9. Funções Agregadoras e Agrupamento

Funções agregadoras são usadas para consolidar (resumir, sumarizar) informação de múltiplos registros em um único registro consolidado. **Agrupamento** é usado para criar subgrupos de registros antes de consolidá-los. Agrupamento e funções agregadoras são necessários em muitas aplicações de bancos de dados, e aqui faremos um estudo introdutório de seu uso em SQL, através de exemplos.

Existem várias funções agregadoras, como COUNT, SUM, MAX, MIN, AVG.

COUNT retorna o número de registros ou de valores que atendam a uma condição de seleção.

SUM, MAX, MIN e AVG podem ser aplicados a um conjunto ou multiconjunto de valores numéricos e retornam, respectivamente, a soma, o maior valor, o menor valor e a média aritmética dos valores envolvidos na consulta. Essas funções podem ser usadas em uma cláusula **SELECT** ou em uma cláusula **HAVING** (que estudaremos logo mais).

MAX e MIN também podem ser usados com valores de tipos não-numéricos, desde que seja possível determinar uma ordem entre os valores, estabelecendo a sequência ordenada entre eles. Por exemplo, datas e horas permitem esse tipo de ordenação e, portanto, podem ser usados com as funções MAX e MIN.

A seguir estudaremos exemplos e detalhes dessas funções do SQL e sua aplicação no SQL Server.

Consulta 25: Calcule a soma de salários de todos os empregados, o valor do maior e do menor salários e a média salarial da companhia.

```
SELECT
    SUM (Salario) as Total,
    MAX (Salario) as Maior,
    MIN (Salario) Menor,
    AVG (Salario) Média
FROM
    Empresa.Empregado;
```

	Total	Maior	Menor	Média
1	359000.00	55000.00	20000.00	32636.363636

Caso desejássemos os mesmos dados, mas para um departamento específico, poderíamos usar a consulta 26, onde os registros de empregados são selecionados por uma cláusula WHERE com o critério de seleção, por exemplo, empregados que estão alocados no departamento ‘Pesquisa’ :

Consulta 26: Calcule a soma de salários, o valor do maior e do menor salários e a média salarial dos empregados do departamento ‘Pesquisa’:

```
SELECT
    SUM (Salario) as Total,
    MAX (Salario) as Maior,
    MIN (Salario) Menor,
    AVG (Salario) Média
FROM
    (
        Empresa.Empregado E Join Empresa.Departamento D
        On E.NumDept = D.NumDept
    )
Where
    NomeDept = 'Pesquisa';
```

	Total	Maior	Menor	Média
1	133000.00	40000.00	25000.00	33250.000000

Nessa consulta, usou-se o conceito de junção para efetuar a seleção dos registros de empregados cujo departamento seja o de Pesquisa.

Consulta 27: Calcule a quantidade de empregados da companhia.

```
SELECT
    COUNT (*) as 'Quantos empregados'
FROM
    Empresa.Empregado;
```

	Quantos empregados
1	11

Consulta 28: Calcule a quantidade de empregados do departamento ‘Pesquisa’:

```
SELECT
    COUNT (*) as 'Quantos empregados'
FROM
    Empresa.Empregado E,
    Empresa.Departamento D
WHERE
    E.numDept = D.numDept AND nomeDept = 'Pesquisa'
```

	Quantos empregados
1	4

Consulta 29: Calcule a quantidade de diferentes salários na empresa:

```
SELECT
    COUNT (Distinct Salario) as 'Níveis Salariais'
FROM
    Empresa.Empregado;
```

Resultados		Men
Níveis Salariais		
1	8	

Se tivéssemos escrito COUNT(Salario) ao invés de COUNT(Distinct Salario) na consulta 28, valores duplicados não seriam eliminados no resultado. No entanto, quaisquer registros com salário Null não seriam contados. Em geral, valores Null são descartados quando funções agregadoras são aplicadas a qualquer coluna particular.

As consultas 25, 27 e 29 sumarizam uma relação completa ou um subconjunto de registros (consultas 26 e 28) e, portanto, todas elas produzem apenas um registro como resultado. Elas ilustram como funções são aplicadas para recuperar um valor de resumo a partir do banco de dados.

Essas funções também podem ser usadas em condições de seleção usando [consultas aninhadas](#). Podemos especificar uma consulta aninhada correlacionada com uma função agregadora e, então, usar a consulta aninhada na cláusula WHERE da consulta externa.

Por exemplo, a consulta 30 abaixo recupera os nomes de todos os empregados que tem dois ou mais dependentes. Observe que temos uma consulta aninhada (interna) chamada pela cláusula WHERE da consulta externa:

Consulta 30: Recupera o prenome e sobrenome de todos os empregados que tenham 2 ou mais dependentes.

```
SELECT
    Sobrenome, PreNome
FROM
    Empresa.EMPREGADO E
WHERE
    (
        SELECT COUNT (*)
        FROM Empresa.DEPENDENTE D
        WHERE E.NumSegSocial = D.NumSegSocial
    ) >= 2;
```

Seleciona todos os dependentes cujo numSegSocial é igual ao numSegSocial do empregado atualmente analisado pela consulta externa e conta a quantidade de dependentes encontrados desse empregado. Retorna esse valor para o Where

Resultados		Mensagens
	Sobrenome	PreNome
1	Smith	John
2	Wong	Franklin

O valor retornado é comparado e, se for maior ou igual a 2, o registro atualmente analisado da tabela E é selecionado para compor o conjunto de dados resultante.

A consulta interna conta o número de dependentes e retorna o resultado da contagem. Esse resultado é retornado para a cláusula WHERE consulta externa que, por sua vez, compara esse resultado com valor maior ou igual a 2. Sendo assim, a consulta externa recupera nome e sobrenome dos empregados para os quais a consulta interna conta 2 ou mais dependentes.

Se você desejar também exibir a quantidade de dependentes que cada um dos empregados acima possui, terá de repetir a subconsulta (consulta aninhada) na cláusula SELECT, como

vemos abaixo na Consulta 30a. Note que também apresentamos os valores dos NumSegSocial dos empregados envolvidos.

Consulta 30a: Recupera o NumSegSocial, prenome, sobrenome e quantidade de dependentes de todos os empregados que tenham 2 ou mais dependentes.

SELECT

```
Sobrenome, PreNome, (SELECT COUNT (*) as QuantosDependentes
    FROM Empresa.DEPENDENTE D
    WHERE E.NumSegSocial=D.NumSegSocial ) as #Dep
FROM Empresa.EMPREGADO E
WHERE (
    SELECT COUNT (*) as QuantosDependentes
    FROM Empresa.DEPENDENTE D
    WHERE E.NumSegSocial=D.NumSegSocial
) >= 2;
```

O resultado deste SELECT interno é tratado como um campo identificado por #Dep no resultado geral

Resultados		
	Sobrenome	PreNome
1	Smith	John
2	Wong	Franklin

	Sobrenome	PreNome	#Dep
1	Smith	John	3
2	Wong	Franklin	3
3	English	Joyce	0
4	Camargo	Igor	1
5	Assis	Luis	0
6	Narayan	Ramesh	0
7	Kolonsky	Cecilia	0
8	Borg	James	0
9	Wallace	Jennifer	1
10	Jabbar	Ahmad	0
11	Zelaya	Alicia	0

Acima temos o resultado da consulta 30a.

Já ao lado direito temos a consulta sem o uso da cláusula WHERE, para que possamos ver cada funcionário e a sua respectiva quantidade de dependentes. Podemos verificar que a cláusula WHERE realmente selecionou apenas aqueles empregados com dois ou mais dependentes.

[Vídeo adicional – Funções agregadoras](#)

Agrupamento – cláusulas GROUP BY e HAVING

Em muitos casos, desejaremos aplicar as funções agregadoras a subgrupos de registros de uma ou mais tabelas relacionadas, onde os subgrupos são definidos pelos valores de alguns atributos.

Por exemplo, podemos querer encontrar a média salarial de **cada departamento**, ou seja, a média salarial dos empregados agrupados por departamento, ou, então, a quantidade de empregados em **cada projeto**.

Nesses casos, precisamos partitionar a tabela ou relação em subconjuntos sem sobreposição (ou grupos) de registros. Cada grupo (partição) consistirá de registros que tem o mesmo valor do atributo usado para definir o agrupamento. Esse atributo é chamado de **campo de agrupamento**. O campo de agrupamento pode ser formado por atributos compostos, na sequencia hierárquica desejada para esse agrupamento.

Podemos então aplicar a função agregadora a cada grupo independentemente, para produzir a informação que consolida (sumariza) cada grupo.

SQL possui a cláusula GROUP BY para realizar o agrupamento pelo campo desejado. Com ela podemos especificar os campos de agrupamento que, **obrigatoriamente**, também deverão aparecer na cláusula SELECT, de maneira que os valores resultantes da aplicação de cada função agregadora a um agrupamento de registros aparecerão juntamente com os valores dos campos de atributos. A consulta a seguir demonstra esse conceito:

Consulta 31: para cada Departamento, recupera o número do departamento, a quantidade de empregados nele lotados e o salário médio desses empregados do departamento.

SELECT

```
    numDept as 'Dept',
    COUNT (*) as '#Empreg',
    AVG (Salario) as 'Média Salarial'
FROM
    Empresa.EMPREGADO
GROUP BY
    numDept;
```

	Núm. Depto	Quantos Empregados	Média Salarial
1	1	1	55000.000000
2	4	4	30250.000000
3	5	4	33250.000000
4	6	2	25000.000000

Nessa consulta, os registros de EMPREGADO são primeiramente particionados em grupos – cada grupo contendo o mesmo valor do campo de agrupamento numDept. Portanto, cada grupo conterá os empregados que trabalham no mesmo departamento. As funções agregadoras Count() e AVG() são então aplicadas a cada grupo de registros (cada partição).

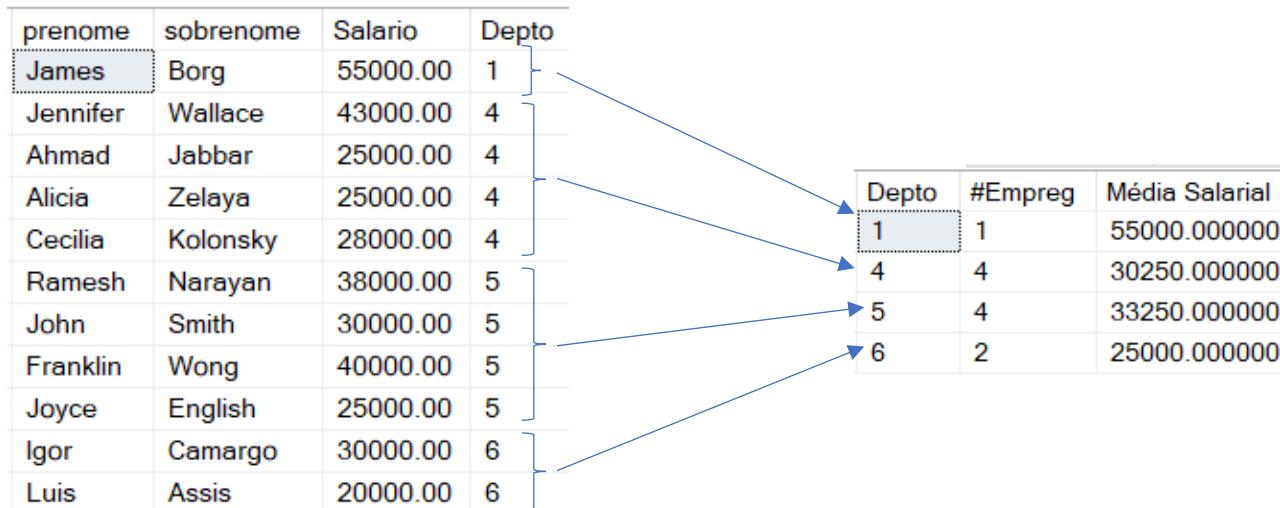
Observe que a cláusula SELECT contém o campo de agrupamento numDept (que deve aparecer, **obrigatoriamente**, no SELECT ao se usar GROUP BY) e as chamadas às funções agregadoras que serão aplicadas a cada grupo.

Se você colocar no GROUP BY um campo da tabela que não deveria ser considerado de agrupamento como, por exemplo, o prenome do empregado, poderá acarretar erro, porque todo campo que aparecer no SELECT quando se usa GROUP BY deve ser uma função agregadora ou os campos usados para agrupamento.

Não faria sentido agrupar por prenome porque, normalmente, os nomes de pessoas (algo bastante **individual**) não identificam **grupos** de registros.

Já o número do departamento, que está previsto num relacionamento com cardinalidade **1:N** (1 Departamento agrupa N Empregados) permite que vários registros de EMPREGADO possam ter o mesmo valor no campo numDept.

A figura abaixo ilustra como o agrupamento funciona na **consulta 31**.



Se, no campo de agrupamento, algum registro tiver o valor NULL, será criado um grupo adicional para todos os registros da tabela que tenham o valor NULL nesse campo. Por exemplo, se a tabela EMPREGADO tivesse um ou mais empregados não associados a nenhum departamento, o campo numDept valeria null e, por isso, haveria um grupo separado para esses registros, com o resultado NULL para numDept.

Também podemos usar GROUP BY em situações de **junção de tabelas**. A próxima consulta demonstra essa situação.

Consulta 32: para cada Projeto, recupere o número do projeto, o nome do projeto e a quantidade de empregados que trabalham nesse projeto.

```
SELECT
    P.numProjeto as #Projeto, nomeProjeto as Projeto, COUNT (*) as Quantos
FROM
    Empresa.PROJETO P, Empresa.TRABALHA_EM T
WHERE
    P.NumProjeto = T.NumProjeto
GROUP BY
    P.numProjeto, nomeProjeto;
```

Podemos reescrever essa consulta usando Inner Join, como vemos na consulta 32a, na qual também ordenamos os resultados pela quantidade decrescente de pessoas trabalhando e, quando estas empatam, pelo campo NumProjeto:

	#Projeto	Projeto	Quantos
1	10	Informatização	4
2	30	Novos Benefícios	3
3	1	ProdutoX	2
4	2	ProdutoY	3
5	3	ProdutoZ	2
6	20	Reorganização	4

Consulta 32a:

```
SELECT
    P.numProjeto as #Projeto, NomeProjeto as Projeto, COUNT (*) as Quantos
FROM
    Empresa.PROJETO P JOIN Empresa.TRABALHA_EM T ON P.numProjeto=T.numProjeto
GROUP BY
    P.numProjeto, nomeProjeto
ORDER BY
    Quantos desc, p.numProjeto;
```

Com esta consulta, podemos prontamente verificar quais projetos tem mais pessoas trabalhando.

	#Projeto	Projeto	Quantos
1	10	Informatização	4
2	20	Reorganização	4
3	2	ProdutoY	3
4	30	Novos Benefícios	3
5	1	ProdutoX	2
6	3	ProdutoZ	2

As consultas 32 e 32a nos mostram como usar uma **condição de junção** (32) ou uma **operação de junção** (32a) em conjunto com GROUP BY. O agrupamento e as funções agregadoras são aplicados **depois** da junção das duas tabelas, sobre todos os registros resultantes da junção.

Embora acima tenhamos aplicado as funções agregadoras a todos os registros do grupo determinado pelos campos de agrupamento, em determinadas situações poderá ser necessário aplicá-las **apenas a registros específicos de um grupo**, que **satisfazem certas condições**.

Por exemplo, suponha que desejamos modificar a consulta 32a de forma que somente projetos com mais de dois empregados apareçam no resultado. SQL fornece a cláusula **HAVING** para **selecionar registros dentro de um conjunto de registros** agrupados pelo GROUP BY.

Somente os grupos que satisfazem a condição do HAVING serão recuperados como resultado da consulta. Esse conceito é mostrado na consulta 33:

Consulta 33: para cada Projeto *no qual mais de dois empregados trabalham*, recupere o número do projeto, o nome do projeto e a quantidade de empregados que nele trabalham.

```
SELECT
    P.numProjeto as #Projeto, nomeProjeto as Projeto, COUNT (*) as Quantos
FROM
    Empresa.PROJETO P JOIN Empresa.TRABALHA_EM T
    ON P.NumProjeto = T.NumProjeto
```

```

GROUP BY
    P.numProjeto, nomeProjeto
HAVING
    Count(*) > 2
ORDER BY
    Quantos desc, P.numProjeto;
  
```

Resultados		Mensagens	
	#Projeto	Projeto	Quantos
1	10	Informatização	4
2	20	Reorganização	4
3	2	ProdutoY	3
4	30	Novos Benefíc	3

Observe que as condições de seleção do WHERE são usadas para selecionar quais registros serão selecionados para aplicação das funções agregadoras. Por outro lado, a condição codificada na cláusula HAVING servem para escolher grupos inteiros, descartando aqueles grupos que não satisfaçam a condição.

#Projeto	Projeto	numSegSocial	NumProjeto	Horas
1	ProdutoX	123456789	1	32.5
1	ProdutoX	453453453	1	20.0
2	ProdutoY	453453453	2	20.0
2	ProdutoY	123456789	2	7.5
2	ProdutoY	333445555	2	10.0
3	ProdutoZ	333445555	3	10.0
3	ProdutoZ	666884444	3	40.0
10	Informatização	333445555	10	10.0
10	Informatização	987987987	10	35.0
10	Informatização	999887777	10	10.0
10	Informatização	512851285	10	20.0
20	Reorganização	494549454	20	25.0
20	Reorganização	333445555	20	10.0
20	Reorganização	888665555	20	40.0
20	Reorganização	987654321	20	15.0
30	Novos Benefíc	987654321	30	20.0
30	Novos Benefíc	987987987	30	5.0
30	Novos Benefíc	999887777	30	30.0

Esses grupos não serão selecionados pela condição do HAVING na consulta 33

Após aplicar a cláusula WHERE mas antes de aplicar HAVING

#Projeto	Projeto	numSegSocial	NumProjeto	Horas
2	ProdutoY	123456789	2	7.5
2	ProdutoY	333445555	2	10.0
2	ProdutoY	453453453	2	20.0
10	Informatização	987987987	10	35.0
10	Informatização	999887777	10	10.0
10	Informatização	333445555	10	10.0
10	Informatização	512851285	10	20.0
20	Reorganização	888665555	20	40.0
20	Reorganização	987654321	20	15.0
20	Reorganização	333445555	20	10.0
20	Reorganização	494549454	20	25.0
30	Novos Benefíc	987987987	30	5.0
30	Novos Benefíc	999887777	30	30.0
30	Novos Benefíc	987654321	30	20.0

#Projeto	Projeto	Quantos
2	ProdutoY	3
10	Informatização	4
20	Reorganização	4
30	Novos Benefíc	3

Após aplicar a condição da cláusula HAVING

Consulta 34: para cada Projeto, recupere o número do projeto, o nome do projeto e a quantidade de empregados do departamento 5 que trabalham nesse projeto.

SELECT

```
P.numProjeto #Projeto, nomeProjeto Projeto, COUNT(*) [Qtos do Depto 5]
FROM
    Empresa.PROJETO P JOIN Empresa.TRABALHA_EM T
        ON P.numProjeto = T.numProjeto
    JOIN Empresa.Empregado E
        On E.numSegSocial = T.numSegSocial
WHERE
    E.numDept = 5
GROUP BY
    P.numProjeto, nomeProjeto
ORDER BY
    [Qtos do Depto 5] desc, P.numProjeto;
```

#Projeto		Projeto	Qtos do Depto 5
1	2	ProdutoY	3
2	1	ProdutoX	2
3	3	ProdutoZ	2
4	10	Informatização	1
5	20	Reorganização	1

Acima nós restringimos os registros no resultado (e, portanto, os registros em cada grupo) para aqueles que satisfaçam a condição especificada na classe WHERE – ou seja, que eles estejam lotados no departamento de número 5.

Observe que temos que ter cautela extra quando duas condições diferentes se aplicam (uma para a função agregadora na cláusula SELECT – COUNT(*) e outra para a função na cláusula HAVING. Por exemplo, suponha que desejamos contar o número total de empregados cujos salários excedam 40000 em cada departamento, mas somente para os departamentos onde trabalham mais de três empregados. Aqui, a condição (SALARIO > 40000) somente se aplica à função COUNT da cláusula SELECT. Imagine que escrevemos a consulta *incorrecta* abaixo:

```
SELECT
    NomeDept, COUNT (*)
FROM
    Empresa.DEPARTAMENTO D, Empresa.Empregado E
WHERE
    D.numDept = E.numDept AND Salario>40000
GROUP BY
    NomeDept
HAVING
    COUNT (*) > 3;
```

#Resultados		Mensagens
	NomeDept	(Nenhum nome de coluna)

Essa consulta está incorreta porque ela seleciona somente departamentos que tenham mais de três empregados em que cada um ganhe mais de 40000. A regra é que a cláusula **WHERE** é executada **primeiramente**, para selecionar registros individuais ou registros de tabelas unidas; a cláusula HAVING é aplicada posteriormente, para selecionar grupos individuais de registros.

Assim sendo, os registros já foram restringidos aos empregados que ganham mais de 40000 **antes** da função COUNT() de HAVING ter sido aplicada.

Uma maneira de escrever essa consulta corretamente é usar uma consulta aninhada, como vemos na consulta 35:

Consulta 35: para cada departamento que tenha mais de três empregados, recupere o número de departamento e a quantidade de empregados que recebem salários maiores de 40000.

```
SELECT D.numDept, COUNT (*) as Quantos
FROM
```

```

Empresa.DEPARTAMENTO D,
Empresa.Empregado E
WHERE
D.numDept = E.numDept and Salario > 40000 AND
D.numDept in
(
    SELECT numDept
    FROM Empresa.EMPREGADO
    GROUP BY numDept
    HAVING COUNT (*) > 3
)
Group by D.numDept

```

	numDept	Quantos
1	4	1

O departamento **4** possui mais de 3 empregados e, ao mesmo tempo, tem empregados ganhando mais de 40000.

Vamos “destrinchar” essa consulta em partes. Primeiramente, vamos trabalhar com a consulta aninhada, fazendo uma pequena modificação para sabermos quantos empregados cada departamento possui:

```

SELECT E.numDept, count(*) as QuantosEmp
FROM Empresa.EMPREGADO E
GROUP BY E.numDept

```

	numDept	QuantosEmp
1	1	1
2	4	4
3	5	4
4	6	2

Aplicando o HAVING para restringir os departamentos do Group By com mais de 3 empregados, teremos o seguinte:

```

SELECT E.numDept, count(*) as QuantosEmp
FROM Empresa.EMPREGADO E
GROUP BY E.numDept
HAVING COUNT (*) > 3

```

	numDept	QuantosEmp
1	4	4
2	5	4

Agora, somente os registros dos departamentos **4** e **5** foram selecionados.

Em seguida, vamos adicionar a busca de dados na tabela de empregados para sabermos os salários:

```

SELECT E.numDept, numSegSocial, Salario
FROM
    Empresa.DEPARTAMENTO D,
    Empresa.Empregado E
WHERE
    E.NumDept = D.NumDept and
    Salario > 40000

```

	numDept	numSegSocial	Salario
1	1	888665555	55000.00
2	4	987654321	43000.00

No resultado acima podemos ver que somente o funcionário 987654321 atende a condição de ganhar mais de 40000 e estar num departamento com mais de 3 funcionários. O departamento 1 somente possui 1 funcionário.

Assim, agora juntamos as duas consultas e agrupamos para termos o resultado desejado:

```
SELECT
```

```

D.numDept, COUNT (*) as Quantos
FROM
    Empresa.DEPARTAMENTO D,
    Empresa.Empregado E
WHERE
    D.numDept = E.numDept and Salario > 40000 AND
    D.numDept in
    (
        SELECT numDept
        FROM Empresa.EMPREGADO
        GROUP BY numDept
        HAVING COUNT (*) > 3
    )
Group by D.numDept

```

	numDept	Quantos
1	4	1

Resumindo, e terminando nossa discussão sobre o comando SELECT, temos abaixo a sua forma geral, com a seis cláusulas que o compõem:

```

SELECT <lista de atributos e funções>
FROM <lista de tabelas | tabelas unidas>
[ WHERE <condição> ]
[ GROUP BY <campo(s) de agrupamento> ]
[ HAVING <condição de agrupamento> ]
[ ORDER BY <lista de atributos> ];

```

[Mais sobre consultas aninhadas \(subconsultas\)](#)

[Subconsultas no SQL Server](#)

[Vídeo adicional – Cláusula With Ties](#)

4.6.10. Comandos de alteração de Esquema

O esquema de um banco de dados é o agrupamento de objetos que, associados, representam no servidor físico a modelagem do banco de dados. Esses objetos são, por exemplo, as tabelas, os campos, as chaves primárias e estrangeiras, índices, triggers, visualizações, funções e procedimentos armazenados, dentre outros.

Conforme o banco de dados vai sendo usado em regime de produção (ou seja, usado para produzir resultados para a empresa), os requisitos da empresa vão evoluindo com a própria dinâmica do mundo dos negócios.

Essa evolução precisa ser acompanhada pelo banco de dados (até mesmo para valorizar o investimento já realizado nesse recurso) que, assim, também muda e evolui.

A evolução do banco de dados se dá através de alterações no esquema, com a adição ou remoção de tabelas, campos, restrições e outros elementos do esquema. Essas alterações podem, em geral, ser feitas enquanto o banco de dados está operacional. No entanto, algumas verificações precisam ser feitas pelo sistema gerenciador de banco de dados e, também, pelo administrador do banco de dados (DBA). Essas verificações têm o objetivo de garantir que as alterações não afetam o restante do banco de dados e não o tornam inconsistente.

Por exemplo:

```

SELECT
    Prenome, Sobrenome
FROM

```

	Prenome	Sobrenome
1	Cecilia	Kolonsky
2	James	Borg

```
Companhia.EMPREGADO
```

```
WHERE
```

```
Super_numSegSocial IS NULL;
```

No comando acima, observe que foi modificado o esquema ao qual pertence a tabela EMPREGADO, de Empresa para COMPANHIA. Usou-se o comando CREATE SCHEMA COMPANHIA para criar o esquema, depois alterou-se as propriedades da tabela Empregado para que ela passasse a fazer parte desse esquema. Essa alteração foi feita com o comando abaixo:

```
ALTER SCHEMA COMPANHIA TRANSFER Empresa.EMPREGADO
```

Isso também foi feito para as demais tabelas desse banco (Empresa), de forma que todas ficaram agrupadas em outro esquema que não o Empresa, e sim o Companhia.

4.6.11. Comando Drop

Este comando pode ser usado para remover elementos nomeados do esquema, tais como tabelas, domínios (tipos definidos pelo desenvolvedor) ou restrições (chaves estrangeiras, defaults, etc.).

Existem duas opções de comportamento no comando DROP: CASCADE e RESTRICT.

Por exemplo, para remover o esquema Companhia e todas as suas tabelas, relacionamentos e outros elementos, a opção CASCADE seria usada como segue abaixo:

```
DROP SCHEMA Companhia CASCADE;
```

O comportamento cascade existe na SQL padrão, é implementado em Oracle (por exemplo) mas o Sql Server não o implementa. SQL Server somente atua com DROP SCHEMA usando a opção RESTRICT, que somente elimina o esquema se ele estiver vazio. Você deveria, nesse caso, eliminar manualmente (usando DROP) cada objeto dentro do esquema antes de apagar o próprio esquema.

Algumas soluções programadas podem ser encontradas na Internet sobre como desenvolver um procedimento armazenado que descobre quais os objetos de um esquema e os elimina, antes de eliminar o próprio esquema. Assim teríamos algo mais automatizado para eliminar um esquema no SQL Server. O link ao lado tem exemplos: [procedure para DROP SCHEMA Cascade](#).

No entanto, depois que o banco de dados está em regime de produção, será raro ter que eliminá-lo totalmente. Assim, essa situação de DROP SCHEMA não será muito usada na prática.

Se você não precisa mais de uma tabela no esquema, a tabela e suas definições podem ser eliminados pelo uso do comando DROP TABLE. Por exemplo, se não mais precisarmos da tabela DEPENDENTE do esquema Companhia, podemos executar o comando abaixo no Sql Server Management Studio:

```
drop table companhia.DEPENDENTE
```

Essa tabela somente será eliminada se não é referenciada por nenhuma restrição (constraint) como, por exemplo, chaves estrangeiras. Se a opção cascade tiver sido implementada no servidor de banco de dados que está sendo usado, todas as tabelas dependentes dessa serão também eliminadas, bem como as constraints que a referenciam, visualizações, e outros elementos associados.

Também se pode usar o comando DROP para eliminar outros tipos de elementos nomeados do esquema, como constraints e domínios.

4.6.12. Comando Alter

Este comando permite alterar definições dos elementos presentes no esquema de banco de dados. Pode-se usá-lo para adicionar ou eliminar campos em tabelas já existentes, alterar a definição de um campo (tipo, tamanho), adicionar ou eliminar constraints, por exemplo.

Já vimos um [exemplo](#) de Alter Table para adicionar chaves estrangeiras a uma tabela.

No SQL Server, se quisermos adicionar um campo a uma tabela, usamos ALTER TABLE com ADD, como abaixo:

```
Alter Table Companhia.Empregado ADD Job Varchar(12) NULL;
```

Acima, estamos adicionando o campo Job à tabela Empregado, e o tipo desse campo será Varchar(12). A definição NULL permitirá que valores nulos sejam colocados nesse campo. Na verdade, se a tabela já tiver dados, será importante colocar essa definição, pois não teremos valores armazenados nesse campo. NOT NULL, portanto, não é permitida neste caso.

Você ainda deverá armazenar um valor de Job para cada registro individual de EMPREGADO. Outra alternativa seria colocar uma definição de valor Default como, por exemplo:

```
Alter Table Companhia.Empregado DROP Column Job ;
```

```
Alter Table Companhia.Empregado ADD Job Varchar(12) Not Null  
Constraint DF_Job Default ('Nada') With Values;
```

	Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno	Job
1	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.00	333445555	5	Nada
2	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.00	888665555	5	Nada
3	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.00	333445555	5	Nada
4	Igor	N	Camargo	494549454	1985-10-26	53 Dona Libânia,Campinas,SP	M	30000.00	888665555	6	Nada
5	Luis	F	Assis	512851285	1994-06-21	37 Reg Feijó, Campinas, SP	M	20000.00	888665555	6	Nada
6	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000.00	333445555	5	Nada
7	Cecilia	F	Kolonsky	677678989	1960-04-05	6357 Windy Lane, Katy, TX	F	28000.00	NULL	4	Nada
8	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.00	NULL	1	Nada
9	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.00	888665555	4	Nada
10	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.00	987654321	4	Nada
11	Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000.00	987654321	4	Nada

Acima, demos o nome DF_JOB para a constraint de valor default do campo Job. Isso fará com que todo registro já existente passe a ter o campo Job valendo 'Nada'. Agora podemos tirar esse default, se assim desejarmos:

```
Alter Table Companhia.Empregado Drop Constraint DF_Job;
```

Agora, qualquer novo registro que for incluído com o campo Job valendo Null não será aceito.

Por falar em inclusão de novos registros, quando tempos um campo autoincremento, seu valor vai sendo incrementado a cada novo registro. Esse valor é controlado pelo sistema gerenciador de banco de dados. Depois de incluir um registro em qualquer tabela, como podemos fazer para saber qual foi esse valor? Para isso existe a variável @@Identity do SQL Server que pode ser retornada com um Select, como vemos abaixo:

```
Select @@Identity as UltimoId
```

4.7. Resumo da Sintaxe de SQL

```

CREATE TABLE <table name>
( <column name> <column type> [ <attribute constraint> ]
  { , <column name> <column type> [ <attribute constraint> ] }
  [ <table constraint> { , <table constraint> } ]
)

DROP TABLE <table name>

ALTER TABLE <table name> ADD <column name> <column type>

SELECT
    [ DISTINCT ] <attribute list>
FROM
    ( <table name> { <alias> } | <joined table> )
    { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]

<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ]
<column name> | * ) ) )
{ , ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) } ) )
<grouping attributes> ::= <column name> { , <column name> }

<order> ::= ( ASC | DESC )

INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { ,
<constant value> } ) }
| <select statement> )

DELETE FROM <table name>
[ WHERE <selection condition> ]

UPDATE <table name>
    SET <column name> = <value expression>
        { , <column name> = <value expression> }
[ WHERE <selection condition> ]

CREATE [ UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ]

DROP INDEX <index name>

```

Acima temos um resumo com os recursos que estudamos neste semestre, mas que não esgotam todos os elementos existentes na SQL.

Usamos notação BNF, onde símbolos não-terminais são apresentados entre <...>, partes opcionais são apresentados entre [...], repetições são apresentadas entre {...}, e alternativas são apresentadas entre parênteses e | (... | ... | ...).

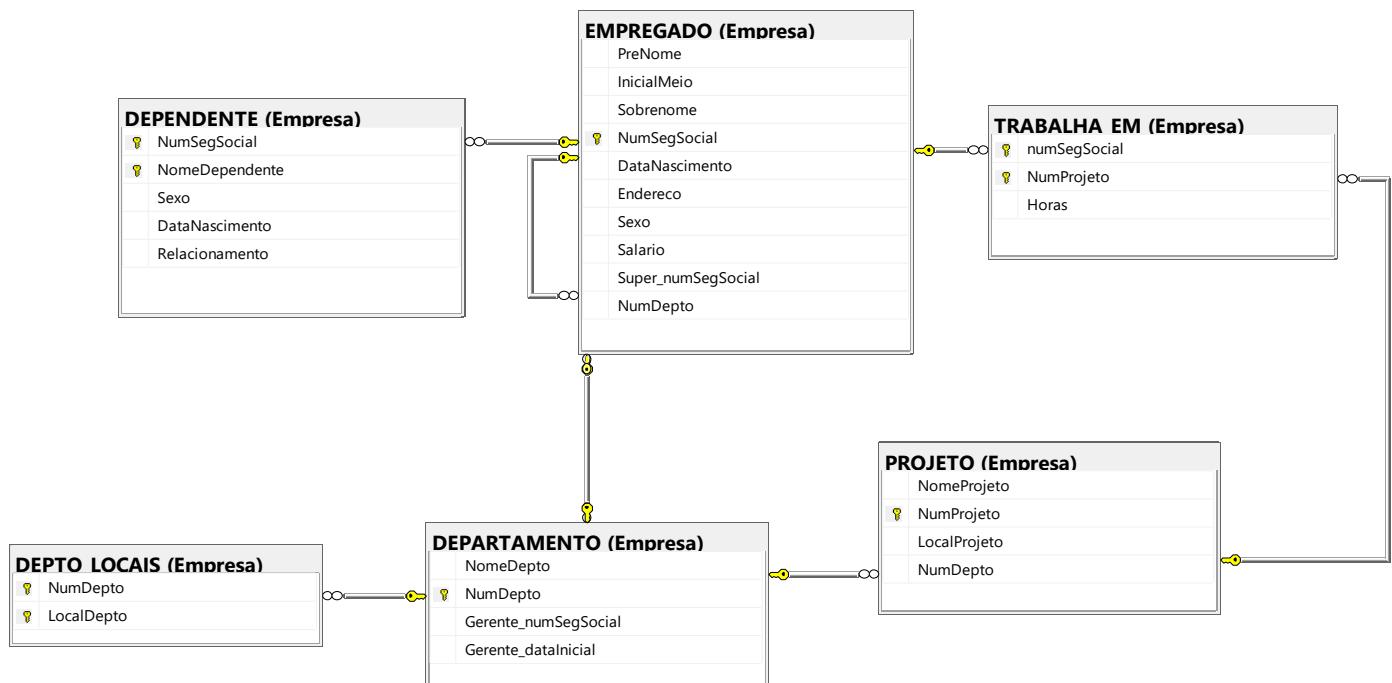
Questões de Revisão

Provenientes do livro Fundamental of Data Base Management Systems, de Ramez Elmasri e Shamkant B. Navathe

1. Descreva as seis cláusulas da sintaxe de uma consulta SQL. Mostre quais tipos de elementos podem ser especificados em cada uma das seis cláusulas. Quais dessas seis cláusulas são obrigatórias e quais são opcionais?
2. Explique como NULLs são tratados em operadores de comparação em uma consulta SQL? Como eles são tratados se existirem em atributos de agrupamento?
3. Discuta como cada um dos seguintes elementos é usado em SQL e discuta as várias opções de cada elemento, bem como para o que é usado:
 1. Consultas aninhadas
 2. Tabelas unidas e junções externas
 3. Funções agregadoras e agrupamento
 4. Comandos de alteração de esquema

Exercícios

4. Levando em consideração a figura abaixo, que se refere ao diagrama do esquema do banco de dados relacional Empresa, especifique as consultas solicitadas. Mostre os resultados de cada consulta se ela for aplicada ao banco de dados físico que estamos usando em nosso estudo.



1. Para cada departamento cujo salário médio dos empregados é maior que 30000, recupere o nome do departamento e a quantidade de empregados que trabalham nesse departamento.
2. Suponha que desejamos a quantidade de empregados do sexo masculino em cada departamento, que receba um salário maior que 30000. Qual consulta deveria ser escrita e qual seu resultado?
3. Usando o conceito de consultas aninhadas e demais assuntos estudados neste capítulo, escreva a consulta que recupera os nomes de todos os empregados que trabalham no departamento que possui o empregado com o maior salário dentre todos os empregados da companhia.

4. Recupere os nomes de todos os empregados cujo Supervisor do Supervisor tem o NumSegSocial igual a '888665555'.
5. Recupere os nomes dos empregados que recebem ao menos 10000 a mais que o empregado que recebe o menor valor de salário da companhia.
6. Desafio: Recupere o nome do empregado que tem o menor salário e trabalha no departamento que possui a maior média salarial da companhia.

4.8. Características adicionais do SQL

SQL tem uma série de características adicionais que estudaremos logo mais. Estes são os seguintes:

- Visualizações SQL, triggers e regras.
- SQL tem várias técnicas para escrever programas em várias linguagens de programação, que incluem instruções SQL para acessar a uma ou mais bases de dados. Estas incluem SQL embutidas (e dinâmicas), SQL/CLI (Call Level Interface) e o seu predecessor ODBC (Open Data Base Connectivity), e SQL/PSM (Módulos Armazenados Persistentes). Discutiremos também como acessar bases de dados SQL através da programação C# usando o ADO.Net e em Java usando JDBC.
- Cada RDBMS comercial terá, para além dos comandos SQL, um conjunto de comandos para especificar parâmetros físicos de concepção de bases de dados, estruturas de arquivos para relações, e vias de acesso tais como índices. Chamamos a estes comandos uma linguagem de definição de armazenamento (SDL). Versões anteriores de SQL tinha comandos para a criação de índices, mas estes foram removidos da linguagem porque não se encontravam ao nível do esquema conceptual. Muitos sistemas ainda têm os comandos CREATE INDEX; mas requerem um nível de privilégio especial.
- SQL tem comandos de controle de transações. Estes são utilizados para especificar unidades de processamento de bases de dados para efeitos de controlo e recuperação de concorrência (acesso simultâneo por dois ou mais usuários). Discutiremos o conceito de transações e concorrência em mais detalhes.
- SQL tem construções linguísticas para especificar a concessão e revogação de privilégios aos utilizadores. Os privilégios correspondem tipicamente ao direito de utilização de certos comandos SQL para acessar determinadas relações. A cada relação é atribuído um proprietário, e tanto o proprietário como o pessoal de DBA podem conceder a usuários selecionados o privilégio de utilizar uma instrução SQL - como SELECT, INSERT, DELETE ou UPDATE para acessar a relação. Além disso, o pessoal de DBA pode conceder privilégios para criar esquemas, tabelas, ou visualizações a certos usuários. Estes comandos SQL são chamados GRANT e REVOKE.
- SQL tem construções linguísticas para a criação de Triggers (gatilhos). Estes são geralmente referidas como técnicas ativas de bases de dados, uma vez que especificam ações que são automaticamente desencadeadas por eventos como atualizações da bases de dados.
- SQL incorporou muitas características dos modelos orientados para objetos para ter capacidades mais poderosas, levando a sistemas relacionais melhorados conhecidos como objeto-relacional. Capacidades como a criação de atributos estruturados complexos, especificando tipos de dados abstratos (chamados UDTs ou tipos definidos pelo usuário) para atributos e tabelas, criando identificadores de objetos para referenciar as tuplas.
- As bases de dados SQL e relacionais podem interagir com novas tecnologias, tais como XML e OLAP/armazéns de dados.

Nos próximos capítulos passaremos a discutir e praticar esses conceitos.

5. Programação de um aplicativo com acesso direto a Banco de Dados

Vamos criar um aplicativo Windows Forms, usando a linguagem C#, para acessarmos um banco de dados simples e realizarmos operações de manutenção de dados nesse banco.

Como você observará, seu aplicativo será um programa executado em um computador e o servidor de banco de dados será acessado por ele. O servidor de banco de dados, muito possivelmente, será executado em outro computador, talvez até mesmo em uma rede diferente da sua. Para isso, os dois programas terão de se comunicar usando o que chamamos de Arquitetura Cliente/Servidor. Seu programa é o Cliente e o servidor de banco de dados é o programa Servidor.

O Cliente não trata dos dados, apenas faz pedidos de manutenção dos dados ao Servidor, que executa esses pedidos e retorna seus resultados ao Cliente.

Toda a comunicação entre seu aplicativo Cliente e o Servidor se iniciará através de um componente chamado Conexão de Banco de Dados. Enquanto essa conexão for ativa, os dois programas se comunicarão e operações serão realizadas entre as duas pontas da comunicação.

No entanto, para o usuário do seu aplicativo isso não será visível. Para esse usuário, todo o processamento parecerá ser feito no computador que ele está usando.

O material a seguir foi adaptado a partir do artigo do site DevMedia, que pode ser acessado [aqui](#). Há, também, inclusão de outros artigos explanatórios, de Fábio Pires da Bóson Treinamentos.

Cadastro de um Consultório em Windows Forms, com C# e SQL Server

Parte 1

Olá pessoal, começo aqui uma série de artigos criando aplicações simples em Windows Forms usando a linguagem C# e o banco de dados SQL Server. Desta vez iremos criar um sistema de um consultório médico, com cadastro de pacientes, médicos e consultas.

Faço este artigo com base nas videoaulas de Windows Forms de **Luciano Pimenta**, do [Portal Linha de Código](#), mediante autorização do mesmo. Acompanhem o passo-a-passo.

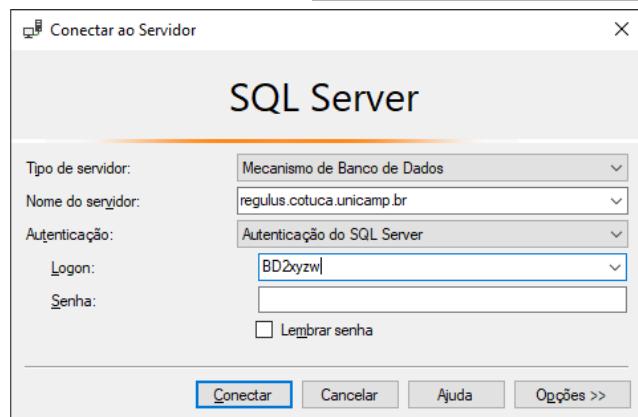
Se estiver fora da rede da escola, precisará usar o OpenVPN que a Unicamp disponibiliza, para poder se conectar à rede do Cotuca e ao servidor de banco de dados acadêmico Regulus.

Abra o SQL Server Management Studio e conecte-se ao servidor Regulus.cotuca.unicamp.br, usando autenticação do Sql Server. Digite seu Logon (por exemplo, BD22209) e senha.

Iremos empregar o banco de dados de cada aluno no servidor de Banco de Dados da **rede acadêmica**. Abra o seu banco de dados e crie uma nova consulta, pressionando o botão [Nova Consulta] ou Ctrl-N.

Na janela de consulta crie um novo esquema, chamado **Cons**, para nele agruparmos as tabelas dessa aplicação. Você pode configurar um filtro nas tabelas, indicando esquema Cons, de forma que apenas as tabelas desse esquema sejam visualizadas no Pesquisador de Objetos.

Para acelerar os estudos, temos aqui o código completo do banco, incluindo a criação de suas três tabelas e respectiva inserção de alguns



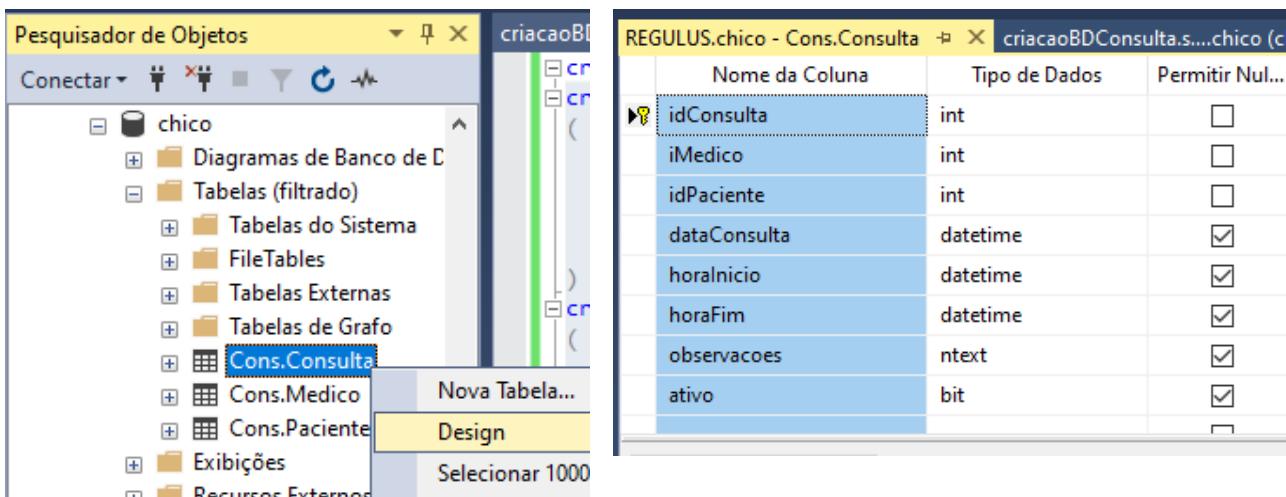
registros. Basta digitá-lo corretamente na janela de consulta do SQL Server Management Studio e executá-lo, para criar as tabelas no esquema Cons.

Na janela de scripts crie um novo **esquema** com o nome **Consultorio**. Após isso crie as tabelas de Pacientes, Médicos e Consultas, conforme mostra a imagem abaixo:

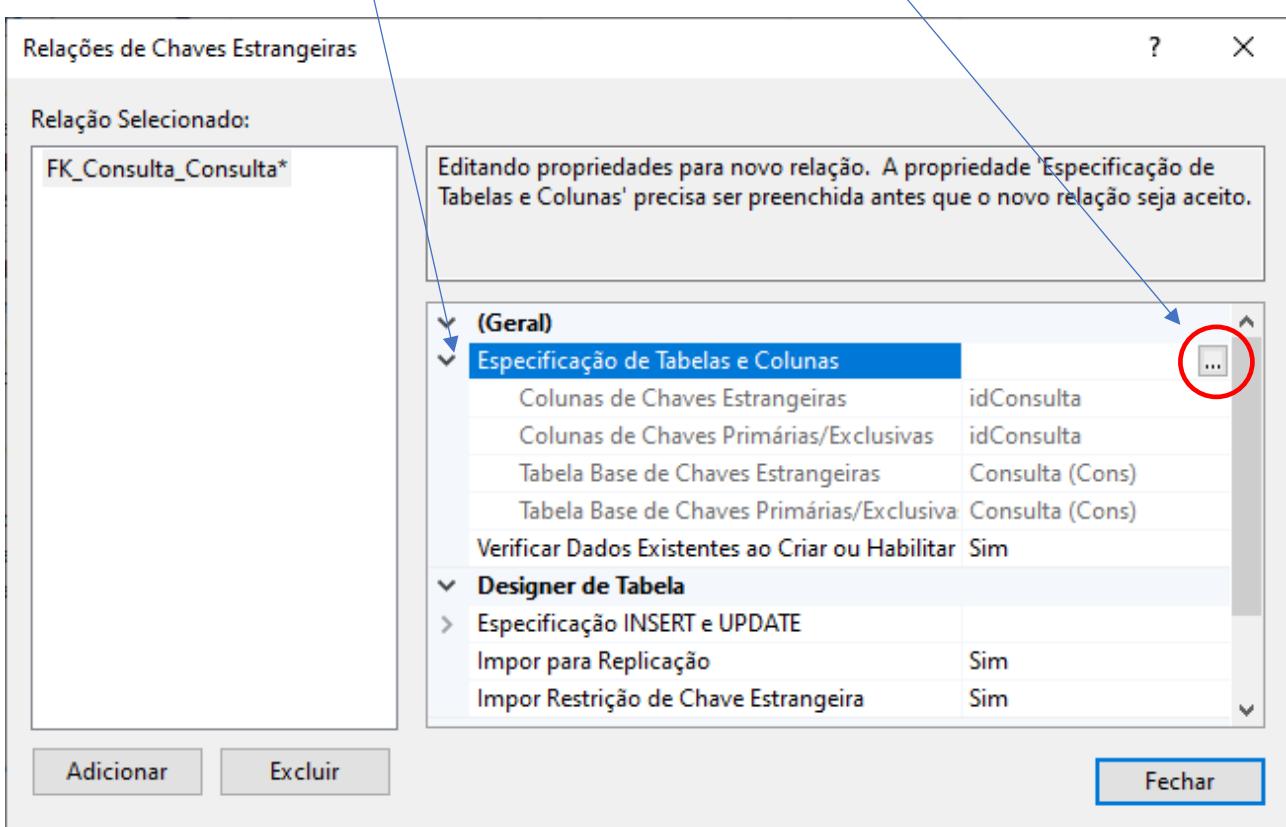
```
create schema Cons
create table Cons.Medico
(
    idMedico      int identity primary key not null,
    nomeMedico    varchar(20),
    sobrenomeMed  varchar(30),
    crm           varchar(10)
)
create table Cons.Paciente
(
    idPaciente    int identity primary key not null,
    nomePaciente  varchar(20),
    sobrenomePac  varchar(30),
    telefone      varchar(20),
    nascimento    date
)
create table Cons.Consulta
(
    idConsulta     int identity primary key not null,
    idMedico       int not null,--foreign key references cons.Medico(idMedico),
    idPaciente     int not null,--foreign key references cons.Paciente(idPaciente),
    dataConsulta   datetime,
    horaInicio     datetime,
    horaFim        datetime,
    observacoes   ntext,
    ativo          bit
)
```

No código acima especificamos (comentados) os relacionamentos entre as tabelas, sendo que a tabela Consulta une as tabelas Medico e Paciente, num relacionamento com cardinalidade 0..N:0..N. Bastaria remover os comentários e ajustar a posição das vírgulas para que os comandos acima já criassem as tabelas e seus relacionamentos. No entanto, é possível usar as ferramentas do SSMS para criar os relacionamentos entre as tabelas, sem o uso da constraint foreign key. Isso é demonstrado nos procedimentos abaixo.

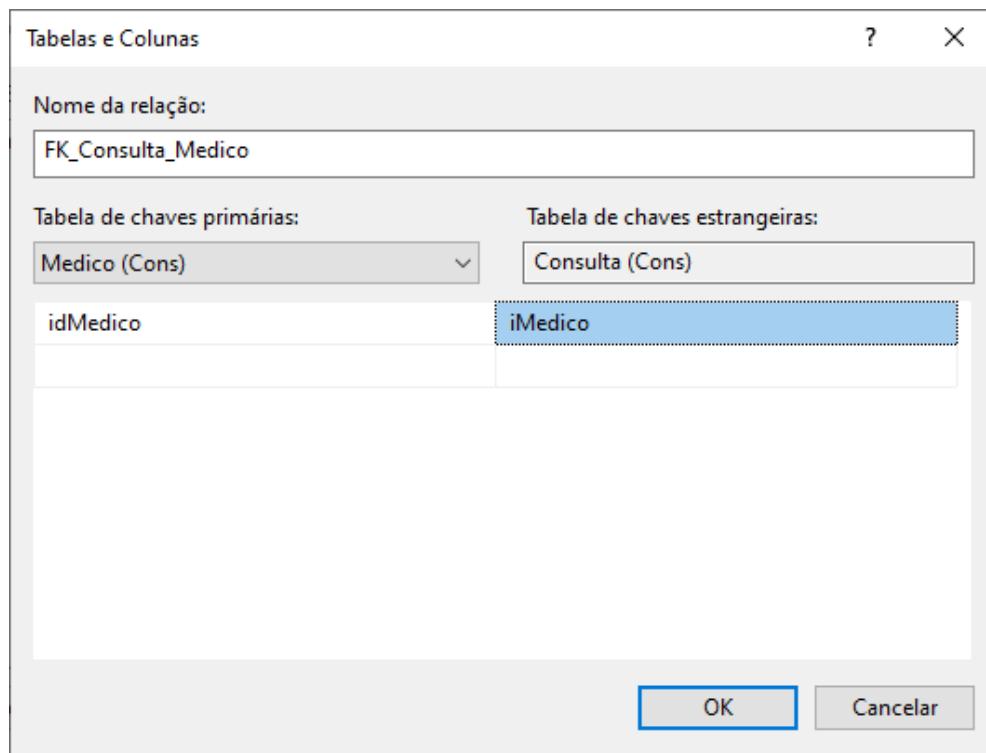
Para isso, após criar as tabelas usando o script acima, acesse o **Solution Explorer**, clique com o botão direito na tabela **Cons.Consulta** recém criada, clique em **Design** e será aberta uma guia com a descrição dessa tabela, como vemos abaixo:



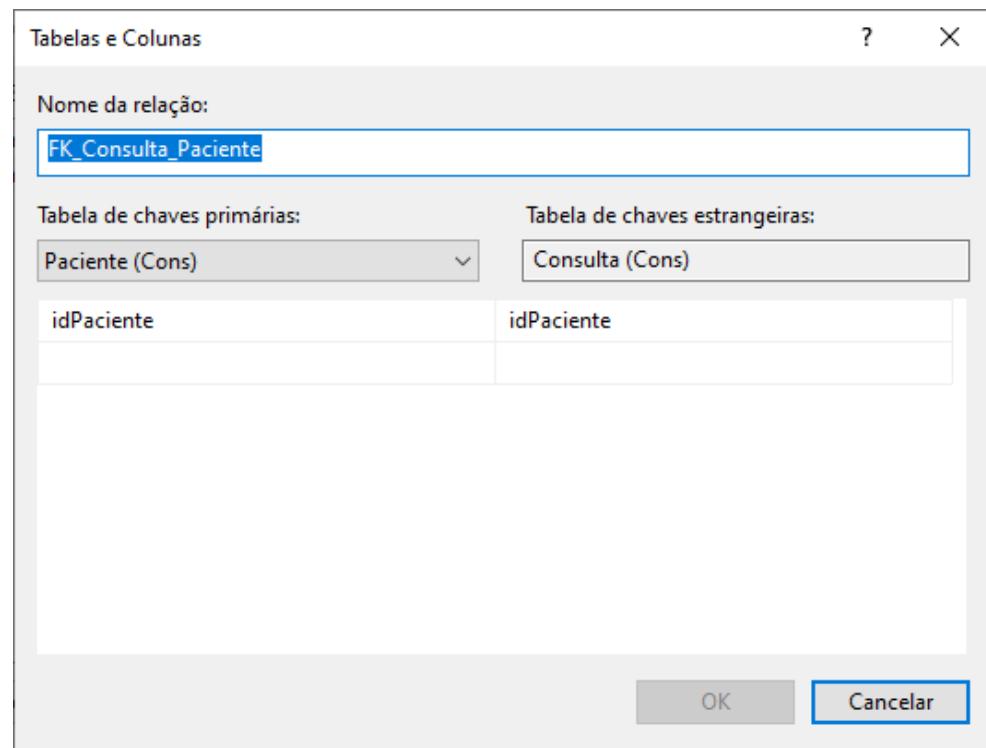
No menu principal do SSMS apareceu uma opção Designer de Tabela. Dentro dessa opção de menu há uma subopção **Relacionamentos...** que deverá ser clicada. Na janela que aparecerá (**Relação de Chaves Estrangeiras**), clique no botão [Adicionar] e, em seguida, expanda a coleção **Especificação de Tabelas e Colunas** e clique no **botão ao lado direito** dessa coleção:



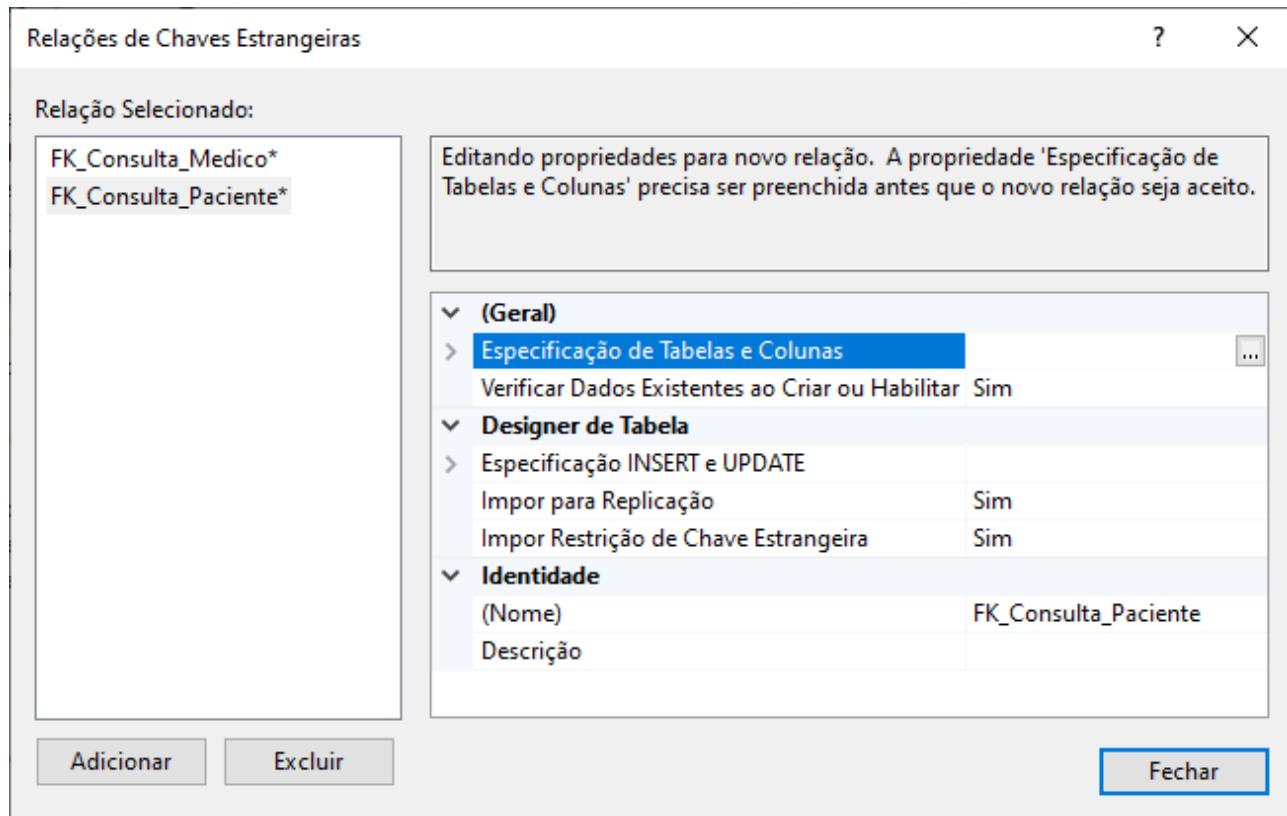
Na janela que abrirá, em **Tabela de chaves primárias**, faça os relacionamentos, escolhendo a tabela Medico e abaixo relate o **IDMedico** da tabela **Medico** com o **IDMedico** da tabela **Consulta**. Pressione o botão [Ok]:



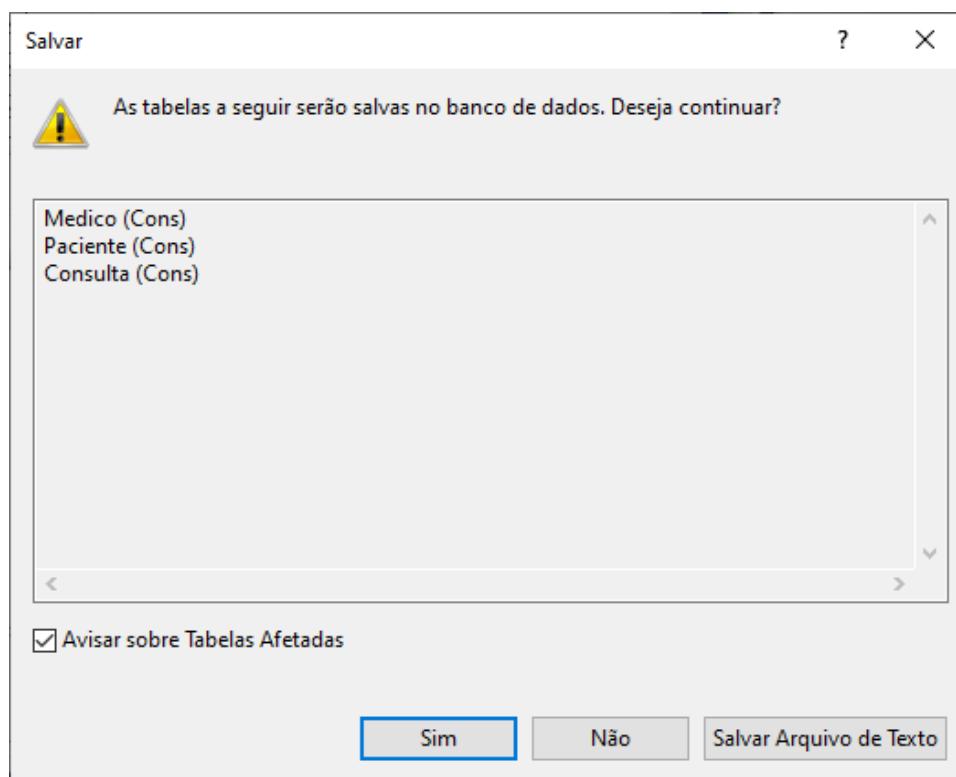
Novamente pressione o botão [Adicionar] na janela Relações de Chaves estrangeiras e repita a associação entre as tabelas Paciente e Consulta:



Após pressionar [Ok], teremos os dois relacionamentos registrados, e a janela **Relações de Chaves Estrangeiras** ficará como abaixo:

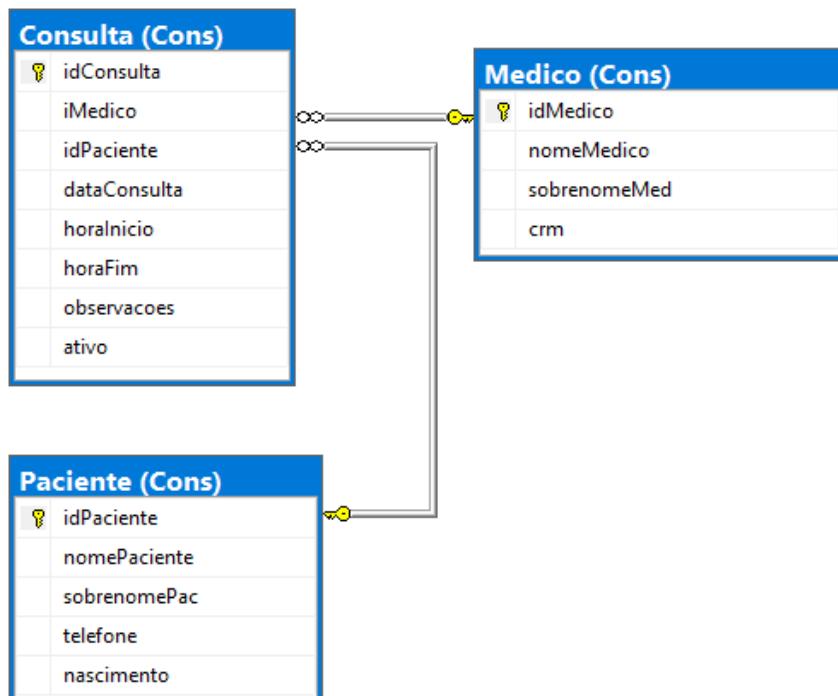


Após isso, clique com o botão direito sobre o título da guia com a descrição da tabela Consulta e selecione a opção Salvar Consulta. Salve a tabela, irá aparecer uma mensagem avisando que as alterações feitas se aplicarão nas demais tabelas, clique em Sim e aguarde.



Você pode criar um diagrama desse esquema, clicando com o botão direito na opção Diagramas de Bancos de Dados e selecionando a subopção **Novo Diagrama de Banco de Dados**. Adicione

as três tabelas acima (Consulta (Cons), Medico (Cons) e Paciente (Cons) e observe que os relacionamentos também já foram criados.



Agora abra o Visual Studio, crie um novo projeto Windows Forms e dê o nome de **apConsultas**.

Configurar seu novo projeto

Aplicativo do Windows Forms (.NET Framework) C# Windows Área de Trabalho

Nome do projeto
apConsultas

Local
G:\Chico\0Cotuca\APOSTILAS\Bancos de Dados\aplicacoes\

Nome da solução
apConsultas

Colocar a solução e o projeto no mesmo diretório

Estrutura
.NET Framework 4.8

Voltar Criar

Será gerado o formulário principal da aplicação. Clique nele, abra a janela **Properties (CTRL+W+P)** altere as seguintes propriedades:

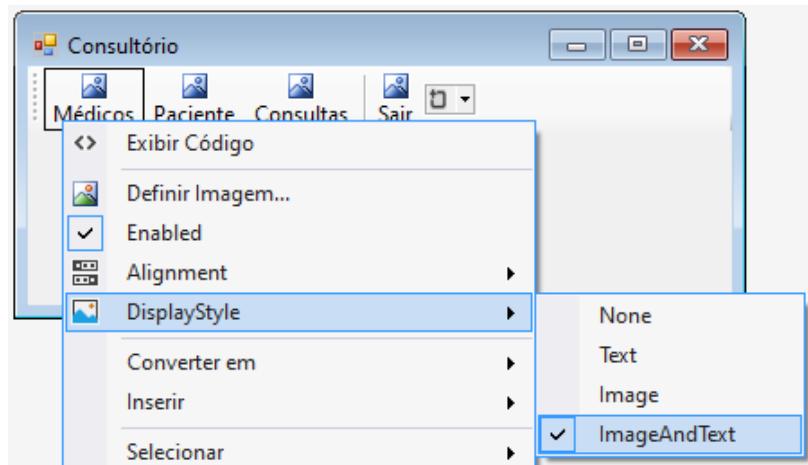
- **Text** – coloque o nome Consultório
- **(Name)** – altere para frmPrincipal
- **WindowState** – altere para Maximized, para abrir sempre maximizado
- **MaximizeBox** – false
- **MinimizeBox** – false – para que só apareça o botão de fechar no form
- **KeyPreview** – true, para ativar o uso do teclado nos eventos do form

Abra a janela **Toolbox** (CTRL+W+X) e arraste da aba **Menus & Barras de Ferramentas** um **ToolStrip**, que nada mais é do que uma barra de controles para seu form. Clique no botão ao lado dele, selecione o controle **Button**, clique com o botão direito nele e clique em **DisplayStyle > ImageAndText**.

Mude o nome do botão para **btnMedicos**, **TextImageRelation** para **ImageAboveText**.

Crie mais dois botões, um separador e um botão, com os mesmos ajustes nas propriedades citadas. Os nomes desses botões são: **btnPacientes**, **btnConsultas**, **btnSair**.

Vamos fazer com que nossa aplicação possa se conectar ao nosso banco de dados. Para isso, precisaremos usar os componentes de acesso a dados, que descrevemos a seguir.



Acesso a Banco de Dados SQL Server: o namespace **SqlClient**

por Fábio dos Reis

O namespace **System.Data.SqlClient** contém diversas classes e outros componentes que permitem realizar a interação com um banco de dados do SQL Server, a partir de uma aplicação em C#, permitindo assim o desenvolvimento de aplicações orientadas a dados. Ele é um Provedor de Dados .NET para o SQL Server (Data Provider).

Este namespace fornece uma grande variedade de funcionalidades, como por exemplo criação de **conexões** a bancos de dados, permitir a **inserção** de dados e a **manipulação** de registros. Ou seja, é um recurso essencial para a construção desse tipo de aplicação.

Para usar as classes disponíveis neste namespace é necessário declará-lo por meio de uma diretiva **using** na seção de namespaces do código da aplicação:

```
using System.Data.SqlClient;
```

Exemplos de classes comumente utilizadas do namespace **SqlClient**:

Classe	Função
SqlCommand	Representa uma instrução T-SQL ou um procedimento armazenado para execução no banco de dados SQL Server.
SqlConnection	Representa uma conexão que será realizada com um banco de dados do SQL Server.
ConnectionStringBuilder	Fornece uma forma simples de criar e gerenciar o conteúdo de strings de conexão usadas pela classe SqlConnection .
SqlDataAdapter	Representa um conjunto de comandos de dados e uma conexão de banco de dados que serão empregados para preencher um DataSet residente na memória e atualizar ou consultar o banco de dados.
SqlDataReader	Fornece uma maneira de ler um fluxo em um único sentido de registros obtidos a partir de um comando Select ou Stored Procedure executados em um banco de dados SQL Server.
SqlException	Exceção gerada quando o SQL Server retorna algum aviso ou erro.
SqlTransaction	Representa uma transação T-SQL a ser feita em um banco de dados do SQL Server.

A lista completa de classes, enumerações e delegates do namespace System.Data.SqlClient pode ser consultada na página da [documentação oficial da Microsoft](#).

Na próxima lição vamos explorar a criação de uma string de conexão a um banco de dados por meio do uso da classe **SqlConnection**.

[Conexão ao banco de dados: a classe SqlConnection](#) - por [Fábio dos Reis](#)

Na lição anterior apresentamos o [namespace SqlClient](#), e agora iremos criar uma string de conexão ao banco de dados usando este namespace.

Criar a string de conexão SQL

Primeiramente precisamos criar a **string de conexão ao banco de dados** desejado. Para isso, após importarmos a classe SqlConnection por meio de uma diretiva *using*, criamos a string de conexão (que chamaremos de strConn), informando a fonte dos dados, nome do banco desejado e tipo de segurança, como podemos ver a seguir:

```
//... Diretivas using ...
using System.Data.SqlClient;

public partial class TelaPrincipal : Form
{
    public string strConn;
    public TelaPrincipal()
    {
        InitializeComponent();
        strConn = "Data Source=Regulus;Initial Catalog=BD22129;Integrated Security=true";
    }
//... Resto do código ...
```

Com a string de conexão criada, podemos agora efetuar a conexão de nossa aplicação a um banco de dados do SQL Server. Veremos como fazer isto na sequência.

Coneectar ao Banco de Dados

Para efetuar a conexão ao banco de dados, criamos um objeto do tipo **SqlConnection** e o inicializamos passando como parâmetro a string de conexão criada:

```
SqlConnection conn = new SqlConnection(strConn);
```

Desta forma, criamos um objeto de nome **conn**, o qual contém a string de conexão **strConn**, declarada no início do código.

Para realizar a conexão ao banco de dados (abrir a conexão), basta agora chamar o método **Open()** do objeto SqlConnection criado:

```
conn.Open();
```

Se a conexão for realizada com sucesso será possível realizar operações no banco de dados indicado, como consultas ou inserções de dados. Veremos como realizar essas operações nas próximas lições deste minicurso.

Após realizar as operações desejadas no banco, é importante fechar a conexão, usando para isso o método **Close()**:

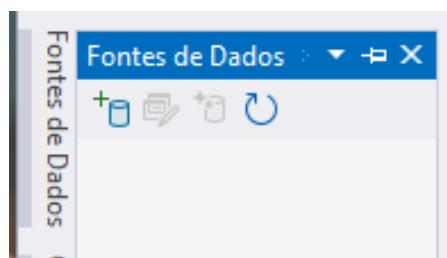
```
conn.Close();
```

Com um objeto de conexão podemos conectar a aplicação ao banco de dados e, através dessa conexão, podemos realizar as operações desejadas no banco, como as operações CRUD (Create, Read, Update, Delete).

Essas operações que vimos nos dois artigos acima são usadas em código programado. Temos também ferramentas no Visual Studio que nos permitem criar uma conexão a um banco de dados de uma forma mais visual, testando também a conexão durante sua criação.

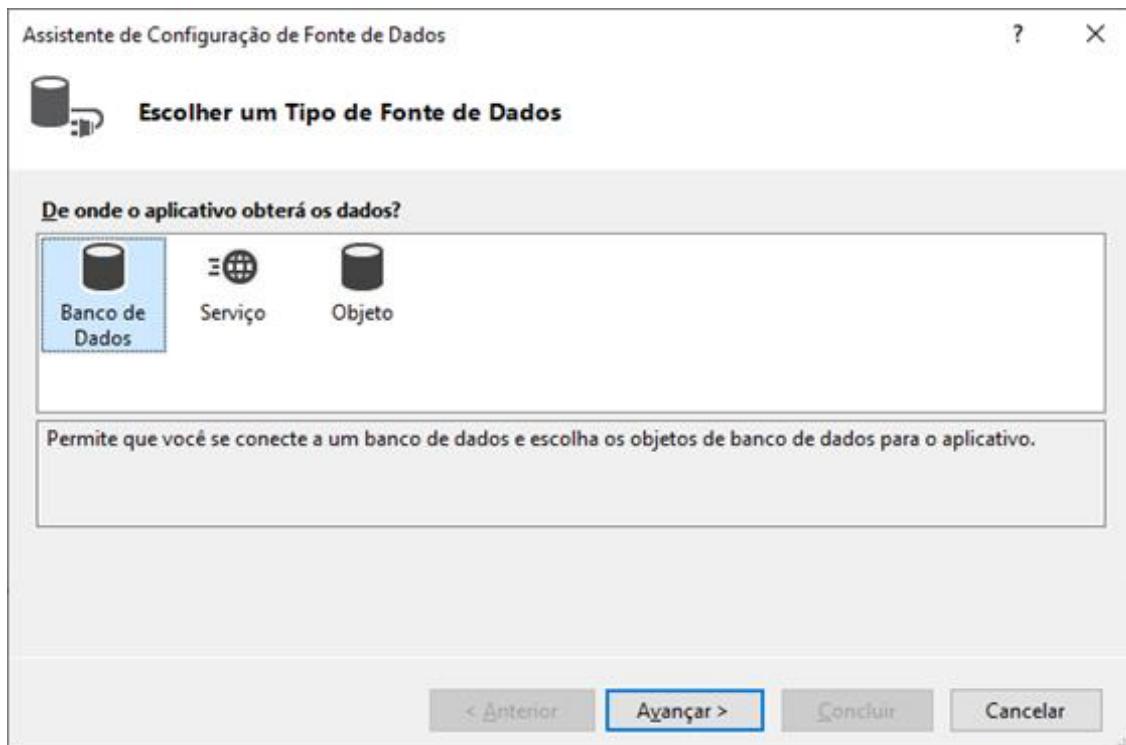
Para usar essas ferramentas, vamos adicionar um novo **Data Source (Fonte de Dados)**. No seu Visual Studio a guia Fontes de Dados já pode estar visível, próxima à Caixas de Ferramentas, como vemos na figura ao lado.

Se ainda não estiver visível, clique no menu Exibir | Outras janelas | Fontes de Dados, para que essa guia fique visível e possa ser usada para criar uma Fonte de Dados que conectará nosso aplicativo ao servidor de banco de dados.

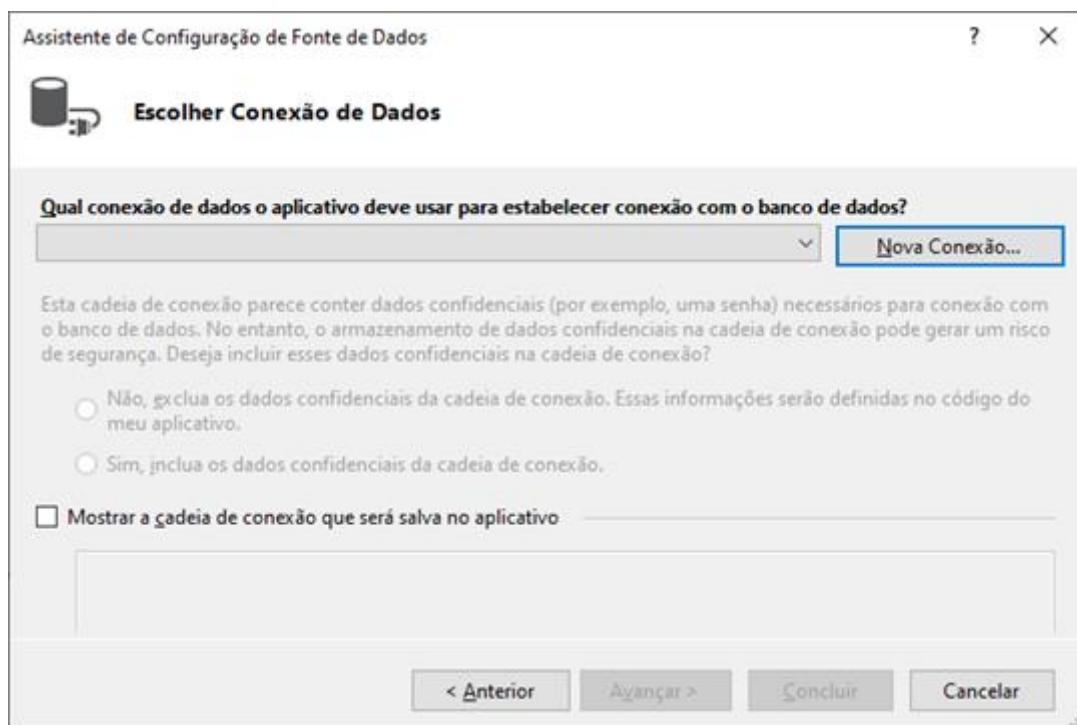


Clique no botão com o símbolo +, que significa Adicionar Fonte de Dados. Aparecerá uma nova janela.

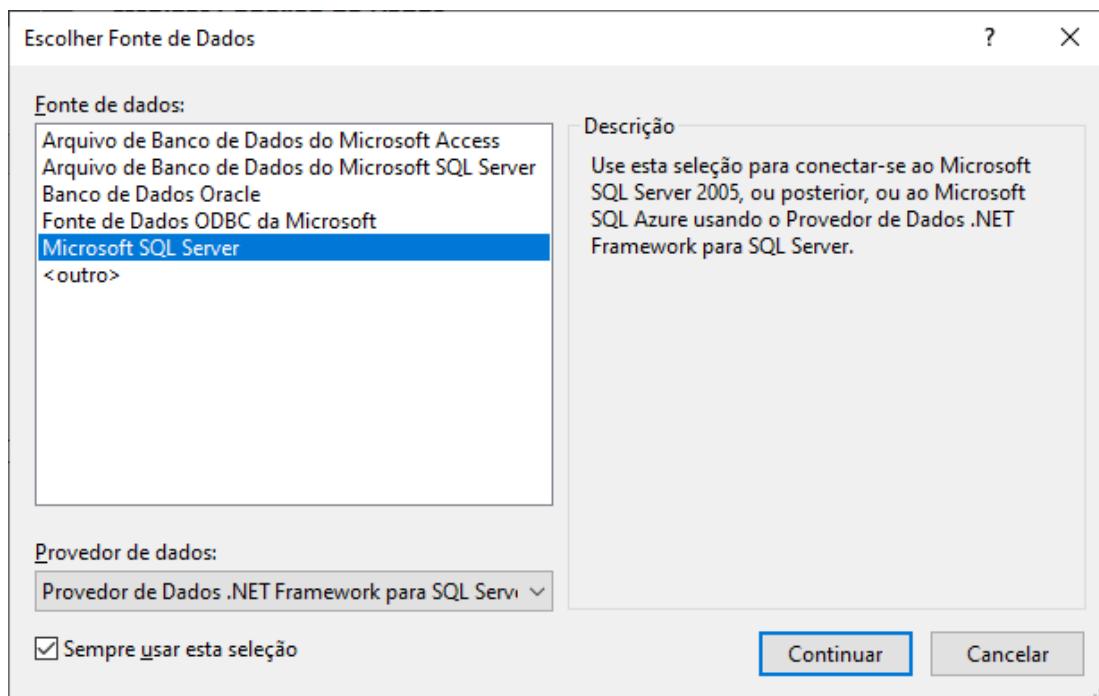
Na janela que aparece, selecione **Banco de Dados** e clique em [Avançar].



Na janela seguinte, clique em **Conjunto de Dados** e, depois, em [Avançar].



Clique em **[Nova Conexão]**, selecione a opção Microsoft SQL Server (atenção: não selecione Arquivo de Banco de Dados do Microsoft SQL Server) e clique em **[Continuar]**.



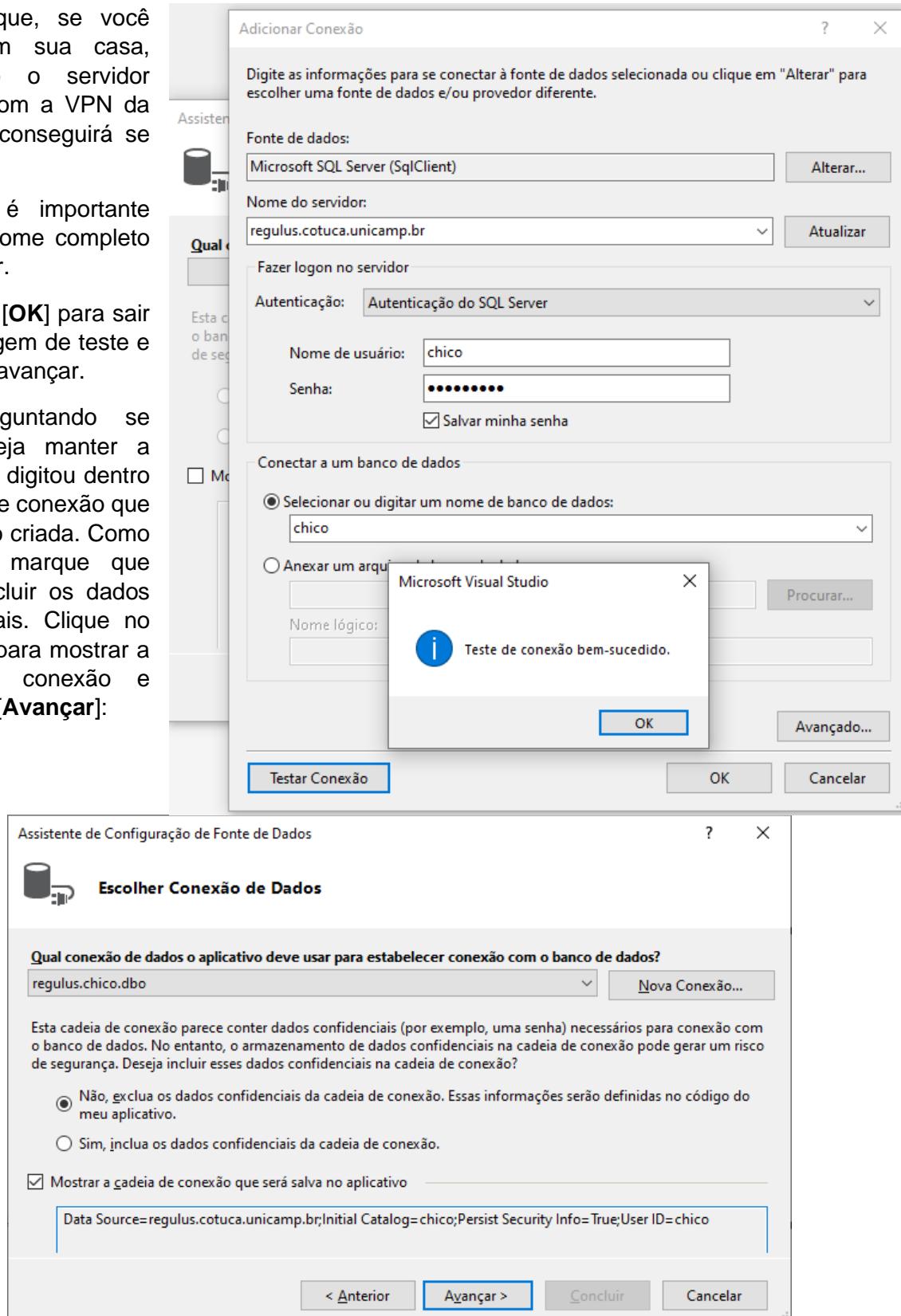
Na janela seguinte, você deverá fornecer as informações para conexão ao seu banco de dados no servidor Regulus, fornecendo o nome do servidor completo, seu nome de usuário, sua senha e indicando o seu banco de dados. Pode clicar no botão **[Testar Conexão]** para verificar se a conexão conseguiu ser estabelecida.

Observe que, se você estiver em sua casa, acessando o servidor Regulus com a VPN da Unicamp, conseguirá se conectar.

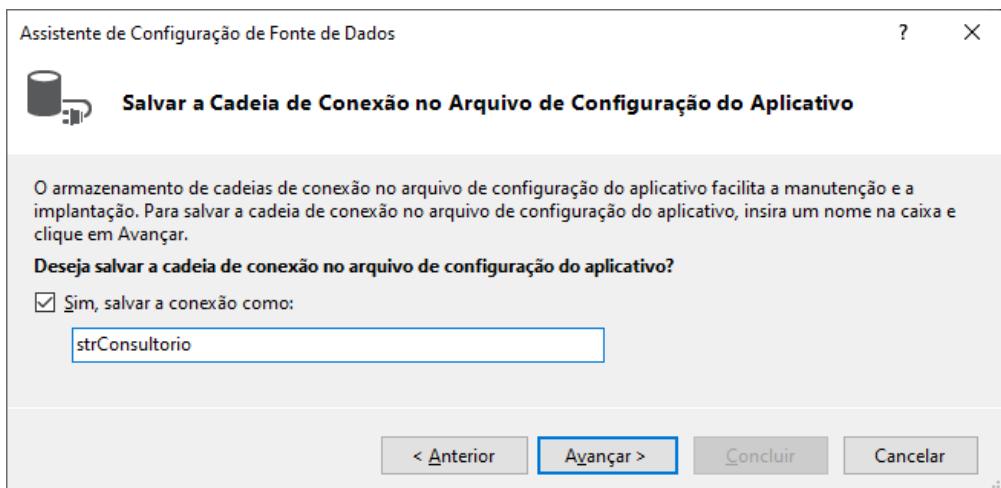
Por isso é importante digitar o nome completo do servidor.

Clique em [OK] para sair da mensagem de teste e [OK] para avançar.

Será perguntando se você deseja manter a senha que digitou dentro da string de conexão que está sendo criada. Como sugestão, marque que deseja excluir os dados confidenciais. Clique no checkbox para mostrar a string de conexão e pressione [Avançar]:



No nome da string de conexão que será solicitado, digite strConsultorio, como vemos abaixo, e clique em [Avançar]:



Assistente de Configuração de Fonte de Dados

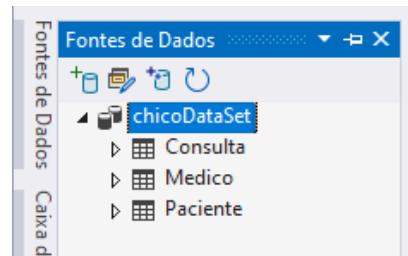
Escolher Objetos do Banco

Quais objetos do banco de dados você gostaria de usar?

- Book_Loans (LIB)
- Borrower (LIB)
- Can_Land (LA)
- Categories
- Cliente
- coConsulta
- coltemReceita
- coMedico
- Consulta (Cons)
- coPaciente
- coReceita
- Course (esc)
- ...

Na próxima tela será perguntado quais objetos você deseja adicionar ao seu **DataSet**, expanda a coleção Tabelas e selecione apenas as tabelas desta aplicação (Consulta (Cons), Medico (Cons) e Paciente (Cons)). Altere o nome do seu **DataSet** se quiser e clique em [Concluir]. Poderá demorar um pouco, caso seu banco de dados no servidor possua muitas tabelas ou sua conexão à Internet seja lenta.

Ficará como vemos na figura à direita:

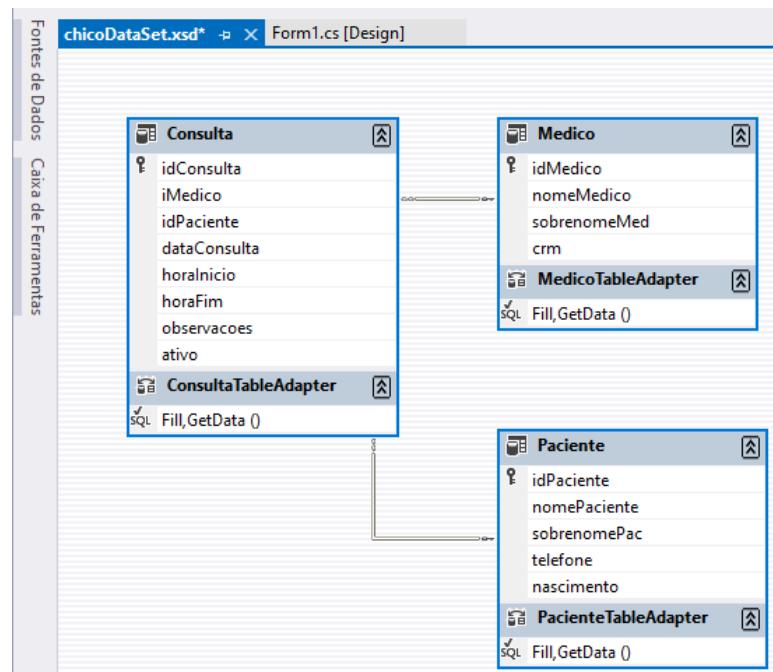


Também será visível uma nova guia na área de código e de design do Visual Studio, mostrando um diagrama das tabelas com classes que as implementam para acesso programado, num mapeamento Relacional → Objeto.

Cada um desses objetos representa nosso banco de dados, com suas tabelas e relacionamentos.

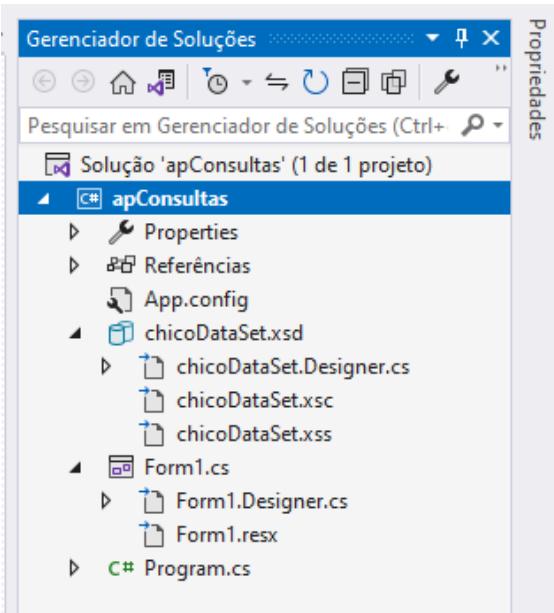
O Visual Studio criou classes que encapsulam atributos e métodos que permitirão ao programa se conectar ao banco de dados, acessar tabelas e seus registros, através de comandos Select, Insert, Update e Delete que estão escritos dentro desses objetos.

Essa ferramenta nos permite, portanto, automatizar muitas das tarefas de acesso a um banco de dados, diminuindo bastante a quantidade de código programável que deveremos codificar.



Assim, podemos nos dedicar à codificação das regras de negócio e outros aspectos da nossa aplicação, pois a interface com o usuário e os comandos de acesso e manutenção das tabelas estarão praticamente prontos.

Se você abrir a janela Solution Explorer (CTRL+W+S) verá que nosso **DataSet** foi criado com sucesso, como vemos ao lado, onde arquivos foram criados com a descrição das tabelas:



No diagrama, clique na tabela Medico. Observe agora a janela de propriedades. Vemos que esse objeto é da classe DataTable.

Essa classe encapsula as operações que acessam banco de dados.

Propriedade	Valor
CaseSensitive	False
Locale	Português (Brasil)
Name	Medico

Clique no campo idMedico e observe a janela de propriedades. É um objeto do tipo DataColumn. Essa classe descreve campos (colunas) de uma tabela.

Na janela de propriedades aparecerão todas as definições desse campo, obtidas no próprio banco de dados pelo Visual Studio, quando criamos o objeto DataSet. Veja que idMedico não permite nulos, é auto-incremento, é do tipo inteiro (Int32), não pode ser modificado (ReadOnly é true) e não pode ser repetido (Unique é true).

Essas configurações estão assim porque esse campo é a chave primária dessa tabela.

Propriedade	Valor
AllowDBNull	False
AutoIncrement	True
AutoIncrementSeed	-1
AutoIncrementStep	-1
Caption	idMedico
DataType	System.Int32
DateTimeMode	UnspecifiedLocal
DefaultValue	<DBNull>
Expression	
MaxLength	-1
Name	idMedico
NullValue	(Throw exception)
ReadOnly	True
Source	idMedico
Unique	True

Depois, observe o campo nomeMedico na janela de propriedades, que é outro DataColumn. Ele permite nulos, não é autoincremento, é do tipo varchar (String) com no máximo 20 caracteres, pode ser modificado na aplicação (ReadOnly é false) e pode ser repetido (Unique é false).

nomeMedico DataColumn	
AllowDBNull	True
AutoIncrement	False
AutoIncrementSeed	0
AutoIncrementStep	1
Caption	nomeMedico
DataType	System.String
DateTimeMode	UnspecifiedLocal
DefaultValue	<DBNull>
Expression	
MaxLength	20
Name	nomeMedico
NullValue	(Throw exception)
ReadOnly	False
Source	nomeMedico
Unique	False

Clique na linha onde está escrito MedicoTableAdapter, e observe na janela de propriedades que temos a string de conexão, código SQL para os comandos Delete, Insert, Select e Update:

Propriedades

MedicoTableAdapter TableAdapter

BaseClass	System.ComponentModel.Component
Connection	strConsultorio (Settings)
Name	strConsultorio (Settings)
Modifier	Internal
ConnectionString	Data Source=regulus.cotuca.unicamp.br;Initial Catalog=chico;Persist Security Info=True;User ID=chico
Provider	System.Data.SqlClient
ConnectionModifier	Internal
DeleteCommand	(DeleteCommand)
CommandText	DELETE FROM [Cons].[Medico] WHERE ([idMedico] = @Original_idMedico) AND (@@IsNull_nomeMedico = 1)
CommandType	Text
Parameters	(Coleção)
GenerateDBDirectMeth	True
InsertCommand	(InsertCommand)
CommandText	INSERT INTO [Cons].[Medico] ([nomeMedico], [sobrenomeMed], [crm]) VALUES (@nomeMedico, @sobrenomeMed, @crm)
CommandType	Text
Parameters	(Coleção)
Modifier	Public
Name	MedicoTableAdapter
SelectCommand	(SelectCommand)
CommandText	SELECT idMedico, nomeMedico, sobrenomeMed, crm FROM Cons.Medico
CommandType	Text
Parameters	(Coleção)
UpdateCommand	(UpdateCommand)
CommandText	UPDATE [Cons].[Medico] SET [nomeMedico] = @nomeMedico, [sobrenomeMed] = @sobrenomeMed, [crm] = @crm WHERE ([idMedico] = @Original_idMedico)
CommandType	Text
Parameters	(Coleção)

Quando criamos o objeto de Fonte de Dados, o Visual Studio preparou todas essas configurações para nós. Sem essa ferramenta, ainda podemos programar todas as funcionalidades que estão descritas e programadas nesse objeto, mas isso levaria mais tempo e estaria sujeito a possíveis erros.

Os comandos definidos nas propriedades DeleteCommand, InsertCommand, SelectCommand e UpdateCommand são o que chamamos de comandos CRUD (Create, Read, Update e Delete).

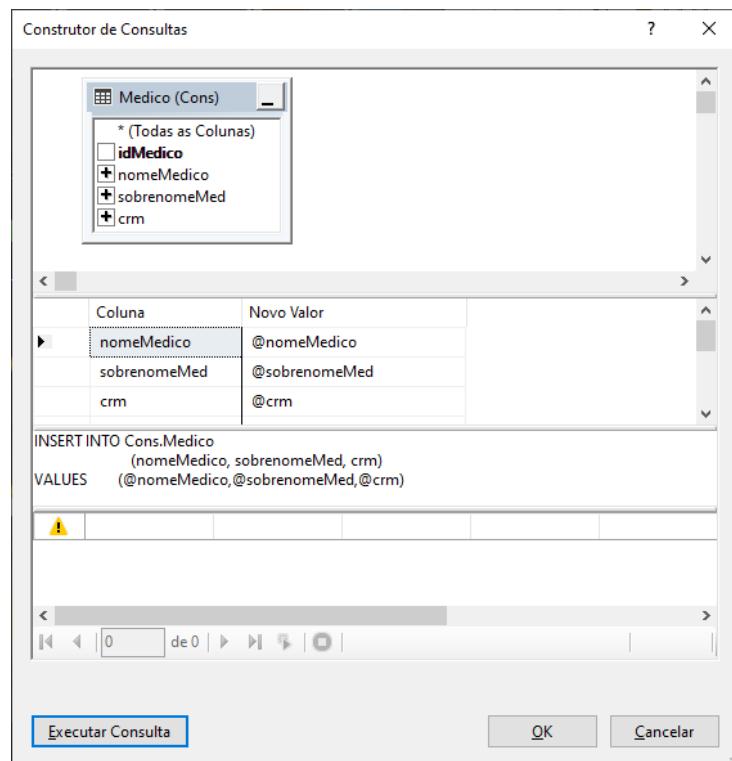
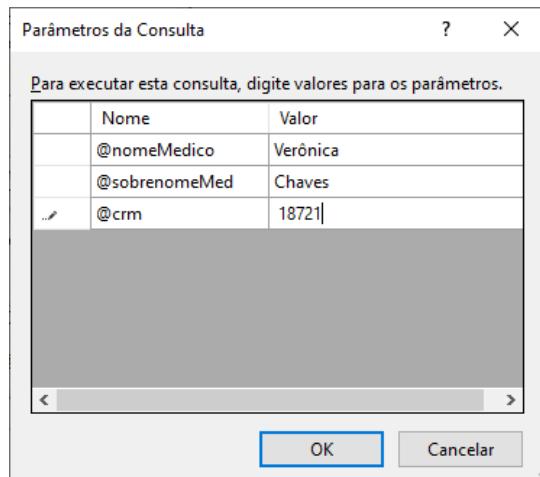
Quando o usuário clicar em botões associados a essas operações, os comandos serão enviados, pela aplicação, ao servidor de banco de dados, usando a conexão que definimos anteriormente. Essa comunicação ocorre através da rede que conecta o computador cliente e o computador servidor.

O servidor de banco de dados avaliará cada comando recebido e, se estiver correto, o executará (na sua própria memória e com seu próprio processador). Assim, nossa aplicação funciona de maneira parecida com o SSMS, que nada mais é do que uma interface na qual digitamos comandos como Delete, Insert, Select e Update para serem executados pelo servidor e seus resultados serem apresentados para nós usuários do SSMS.

Observe agora a propriedade **InsertCommand**. Ela contém o comando Insert para a tabela Medico. Note que na cláusula **Values** existem nomes precedidos pelo caractere @. Esse caractere indica que os nomes que o seguem são parâmetros, ou seja, são como variáveis cujo valor será preenchido pela aplicação e esses valores **substituirão** os parâmetros quando esse comando for **enviado ao servidor** de banco de dados para ser executado.

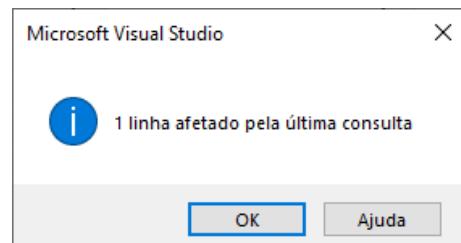
Se você clicar no botão [...] à direita da subpropriedade **CommandText**, abrirá a janela **Construtor de Consultas**, com a qual poderá mudar o comando Insert de acordo com as necessidades específicas da sua aplicação.

Se você clicar em [Executar Consulta], o Visual Studio perguntará quais os valores de nome, sobrenome e CRM que se deseja incluir, como vemos na figura abaixo:



Clicando em [Ok], os parâmetros serão substituídos pelos valores digitados ao lado e o comando Insert será enviado ao servidor.

O servidor recebe esse comando, o analisa e executa. Um novo registro será inserido. Uma mensagem como a abaixo será enviada para o cliente (o Visual Studio, neste momento):



Se você clicar no botão [...] de **CommandText** da propriedade **SelectCommand**, abrirá a janela Construtor de Consultas para o comando Select da tabela de Médicos. Pressionando o botão **[Executar Consulta]**, verá os registros já incluídos nessa tabela. Na figura há 2 registros anteriormente incluídos, além do que acabamos de incluir:

The screenshot shows the 'Consulta' (Query) designer window in Visual Studio. At the top, there is a SQL query:

```
SELECT idMedico, nomeMedico, sobrenomeMed, crm
FROM Cons.Medico
```

Below the query is a grid displaying four rows of data from the 'Medico' table:

	idMedico	nomeMedico	sobrenomeMed	crm
▶	1	Florencio	Gaudencio	1212
▶	2	Joviana	Silva Souza	3343
▶	3	Verônica	Chaves	18721
*	NULL	NULL	NULL	NULL

At the bottom of the grid, there are navigation buttons (back, forward, first, last, etc.) and a status message: "A célula é Somente Leitura." (The cell is read-only).

At the bottom of the window, there are three buttons: 'Executar Consulta' (Execute Query) in a blue box, 'OK', and 'Cancelar' (Cancel).

Como se pode perceber, muita da complexidade de acessar o banco de dados foi delegado para o Visual Studio codificar. Mesmo assim, caso necessário, você poderá configurar as propriedades desses objetos que a ferramenta Fonte de Dados criou, de forma a adequá-los a necessidades específicas da sua aplicação.

Vamos agora começar a criar os formulários da aplicação para tratarmos dos dados de nossas tabelas. Esses formulários usarão as tabelas agrupadas na Fonte de Dados. A interface com o usuário desses formulários usará componentes preparados também na Fonte de Dados e, com isso, poderemos unificar a aparência dos formulários, de forma a facilitar sua utilização pelos usuários.

CADASTRO DE MÉDICOS

No Solution Explorer, clique no nome do projeto (apConsultas) com o botão direito e clique em **Adicionar | Formulário Windows**. Dê o nome de **frmMedico** e clique em **OK**.

Mude sua propriedade **AutoScaleMode** de **Font** para **None**. Mude a propriedade **Font | Size** para **12**.

Altere a propriedade **Text** para Cadastro de Médicos e abra a janela **Fontes de Dados**:

The screenshot shows the 'Fontes de Dados' (Data Sources) window in Visual Studio. On the left, there is a tree view of data sources:

- chicoDataSet
 - Consulta
 - Medico
 - idMedico** (highlighted)
 - TextBox
 - NumericUpDown
 - ComboBox
 - Label
 - LinkLabel
 - ListBox
 - [Nenhum(a)]
- Pac

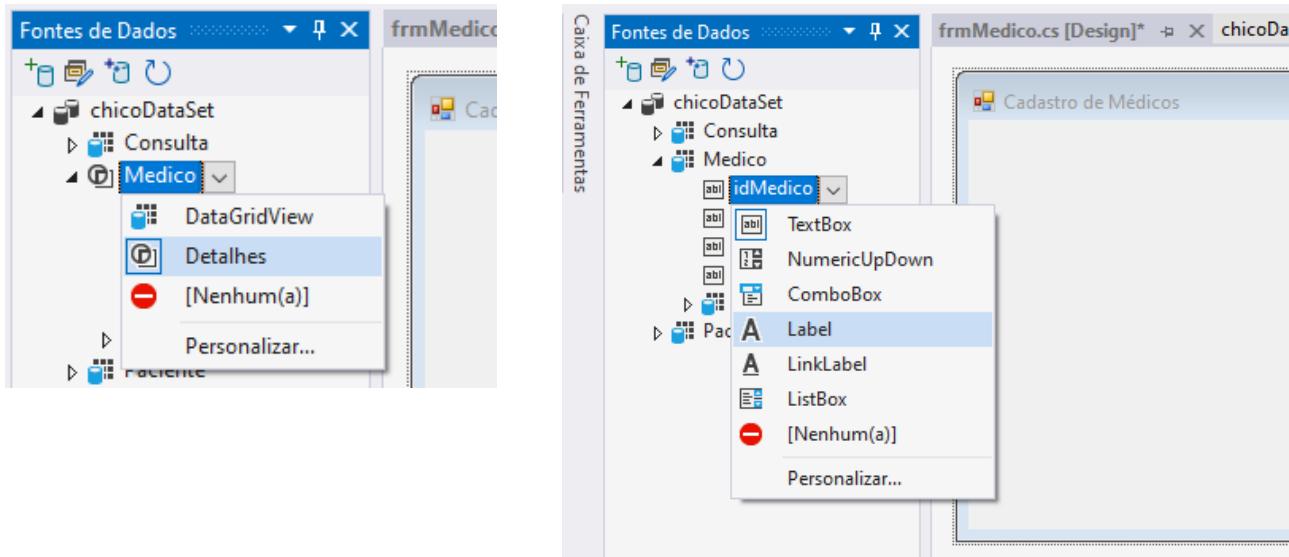
On the right, there is a list of controls corresponding to the selected data source:

- TextBox
- NumericUpDown
- ComboBox
- Label
- LinkLabel
- ListBox
- [Nenhum(a)]

At the bottom, there is a 'Personalizar...' (Customize...) button.

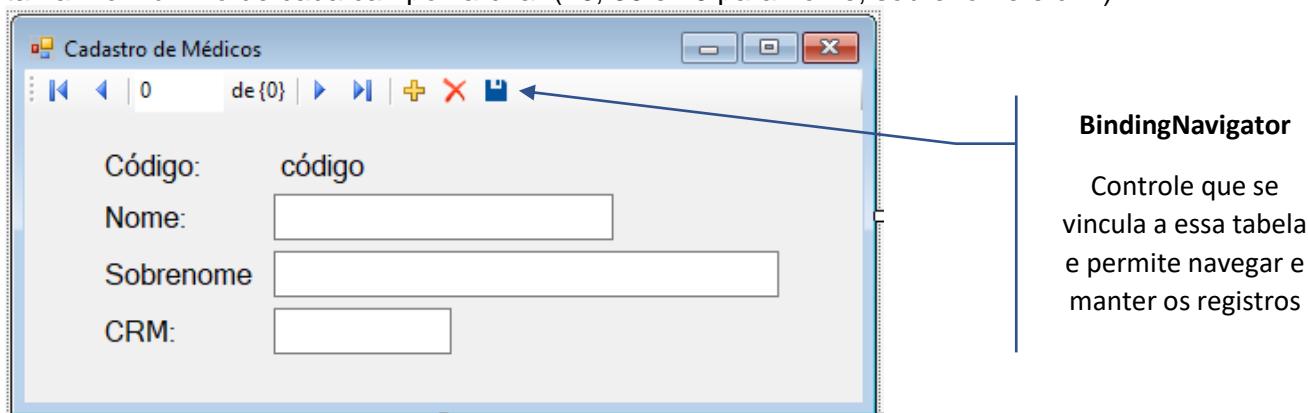
Na janela **Fontes de Dados** que vemos na imagem à esquerda, repare que nossas tabelas trazem os controles adequados a elas, como por exemplo, se você expandir a tabela **Medico**, e clicar em **idMedico** verá que podemos usar um **Textbox**, um **Label**, etc.

Isso é feito automaticamente pelo próprio Visual Studio, ele mapeia os dados que vem do banco SQL e nos mostra quais os controles mais adequados à cada coluna:



Associe a tabela Medico com a opção Detalhes como vemos na figura acima, à esquerda. Em seguida, associe o campo **idMedico** com um controle Label, como mostra a imagem à direita. Como **idMedico** é, no banco de dados, um campo auto-incremento (Identity), não desejamos que o usuário da nossa aplicação possa modificar esse campo, de tal forma que o deixamos apenas para exibição, como um label, no formulário.

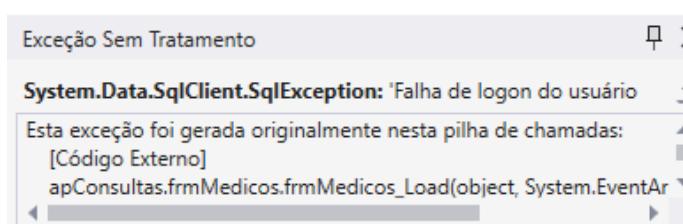
Arraste a tabela **Medico** para nosso form, para que seja criada, automaticamente, a estrutura necessária para tratarmos dados de médicos em nosso form. Faça os ajustes necessários nos campos como mostra a imagem abaixo. Na propriedade **MaxLength** de cada textbox, digite o tamanho máximo de cada campo varchar (20, 30 e 10 para nome, sobrenome e crm).



Agora volte ao formulário principal, clique duas vezes no **btnMedicos** para ir ao evento **btnMedicos_Click** e insira o seguinte código:

```
private void btnMedicos_Click(object sender, EventArgs e)
{
    frmMedicos medico = new frmMedicos();
    medico.ShowDialog();
}
```

O que fiz no código acima foi instanciar o formulário médico e fazer a chamada a ele por meio do método `ShowDialog()`. Agora execute o projeto. Ocorrerá um erro de execução, informando que o logon no servidor não funcionou.



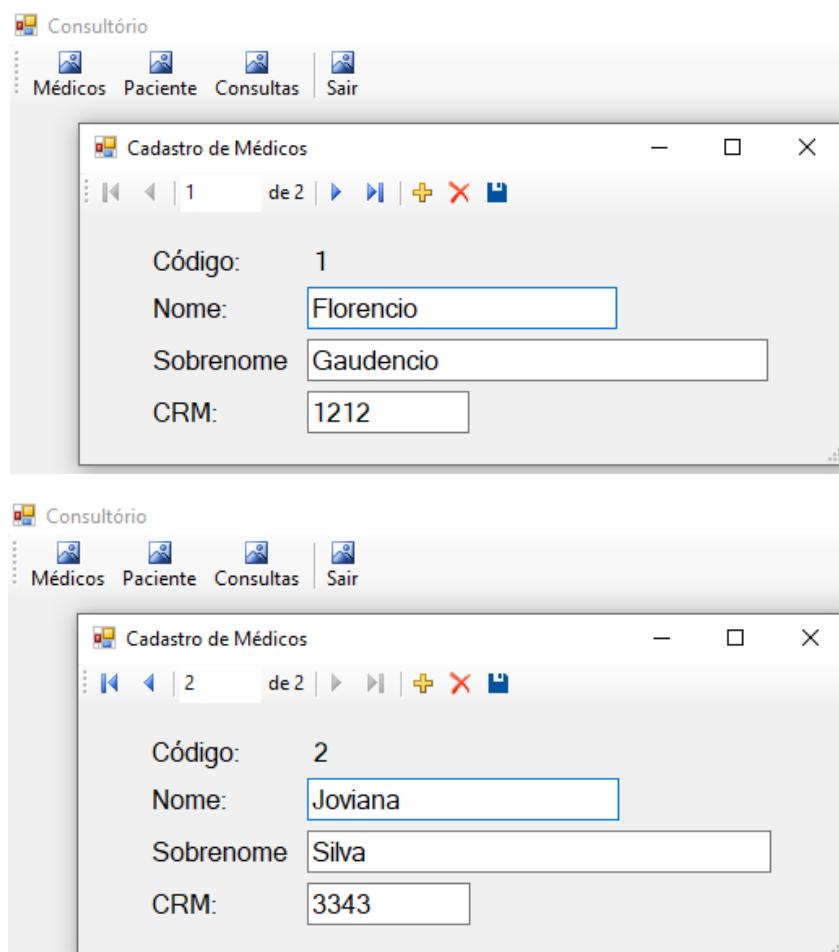
Isso acontece porque nossa aplicação não conseguiu logar no servidor, pois não armazenamos a senha no arquivo de configuração da string de conexão, por uma medida de segurança.

Para resolver isso, podemos pedir a senha no formulário principal usando um textBox e acoplá-la ao tableAdapter de Médicos (um componente criado pelo Designer de Fonte de Dados) ou, neste momento, para ser mais rápido, concatenaremos a senha na própria string de conexão, no evento Load do formulário de Médicos.

O evento Load é executado logo que o formulário é chamado para ser exibido, como fizemos no evento Click do btnMedicos. Nesse evento Load, digite o código abaixo, antes da linha comentada:

```
private void frmMedicos_Load(object sender, EventArgs e)
{
    medicoTableAdapter.Connection.ConnectionString += ";Password=suasenha";
    // TODO: esta linha de código carrega dados na tabela 'chicoDataSet.Medico'.
    // Você pode movê-la ou removê-la conforme necessário.
    this.medicoTableAdapter.Fill(this.chicoDataSet.Medico);
}
```

Fazendo isso e executando novamente a aplicação, ao clicar no botão [Médicos] do formulário principal, aparecerá o formulário de Cadastro de Médicos, após um certo tempo em que se faz a conexão do seu programa com o servidor de banco de dados Regulus:



Experimente inserir, salvar e excluir registros, pare a compilação e compile de novo e note que os registros são salvos no banco e estão disponíveis para posterior consulta. Assim, encerramos aqui a 1ª parte da série de artigos de Windows Forms com acesso à dados.

Créditos a Luciano Pimenta, que fez as videoaulas que deram origem à este artigo e ao Portal [Portal Linha de Código](#), por onde pude baixá-las e estudá-las.

Parte 2

Olá pessoal, continuamos nossa série de artigos criando aplicações simples em Windows Forms usando a linguagem C# e o banco de dados SQL Server. Nesta parte iremos criar o Cadastro de Pacientes e o Cadastro de Consultas e aplicaremos algumas configurações nos formulários.

Abra seu projeto no Visual Studio e abra o modo design do Cadastro de Médicos. Iremos aplicar algumas configurações que serão padronizadas, ou seja, serão aplicadas à todos os forms para que sigam um determinado padrão.

Abra a janela Properties (CTRL+W+P ou F4) e altere as propriedades:

- **FormBorderStyle** - altere para **FixedSingle** para que o form não possa ser redimensionado
- **StartPosition** - coloque **CenterScreen** para que o form abra no meio da tela
- **MaximizeBox** - **false**
- **MinimizeBox** - **false** - para que só apareça o botão de fechar no form
- **KeyPreview** - **true**, para ativar o uso do teclado nos eventos do form

Seu form deverá ficar com propriedades iguais às do form principal.

Agora clique no formulário e aperte F7 para ir à página de código e note que, a partir do momento em que foi adicionado o DataSet e seus demais controles, o Visual Studio automaticamente adicionou também códigos ao nosso form:

```
public partial class frmMedicos : Form
{
    public frmMedicos()
    {
        InitializeComponent();
    }

    private void medicoBindingNavigatorSaveItem_Click(object sender, EventArgs e)
    {
        this.Validate();
        this.medicoBindingSource.EndEdit();
        this.tableAdapterManager.UpdateAll(this.chicoDataSet);
    }

    private void frmMedicos_Load(object sender, EventArgs e)
    {
        medicoTableAdapter.Connection.ConnectionString += ";Password=suasenha";
        // TODO: esta linha de código carrega dados na tabela 'chicoDataSet.Medico'.
        // Você pode movê-la ou removê-la conforme necessário.
        this.medicoTableAdapter.Fill(this.chicoDataSet.Medico);
    }
}
```

Vamos fazer uma verificação simples para que, se o usuário apertar a tecla ESC, feche o formulário. Para isso, na janela propriedades do form frmMedico, vá nos eventos e dê dois cliques no evento KeyDown. Irá se abrir a tela de códigos, nela insira o código abaixo:

```
private void frmMedicos_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape)
    {
        Close();
    }
}
```

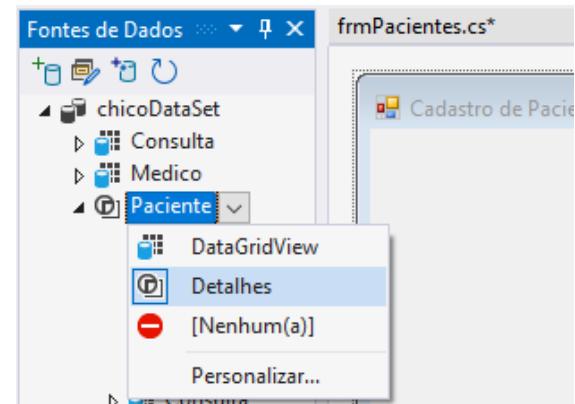
No código acima fiz a verificação se a tecla digitada foi ESC, se foi, automaticamente o form é fechado. Se não, nada acontece. Simples assim.

Cadastro de Pacientes

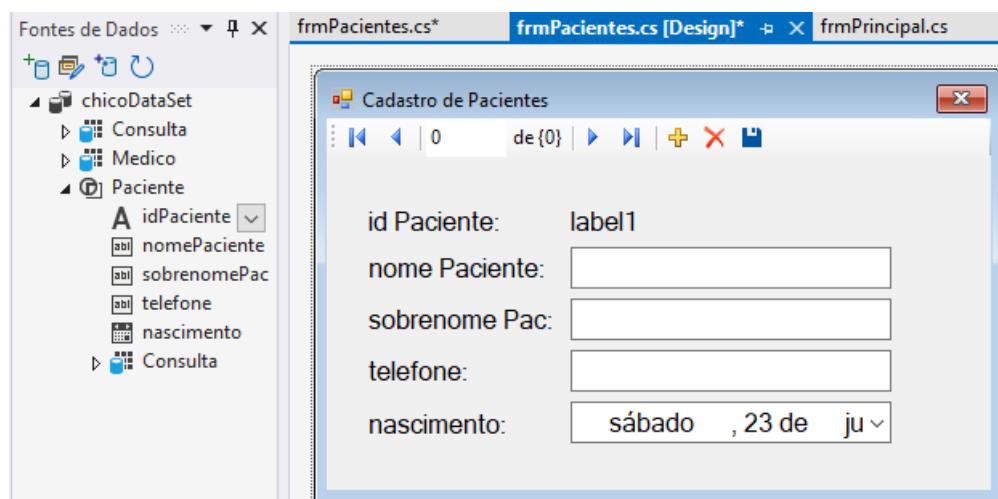
Vamos criar agora um novo form. No Gerenciador de Soluções, clique com o botão direito no nome do projeto, clique em Adicionar | Formulário Windows e dê o nome frmPacientes.

Neste form, aplique as configurações padronizadas que definimos para o formulário de médicos. Mude a propriedade Text para **Cadastro de Pacientes**. Mude AutoScaleMode para None e mude Font|Size para 12. Agora abra o evento KeyDown deste form e aplique o mesmo código do anterior, para que, quando o usuário apertar a tecla ESC, o form feche automaticamente.

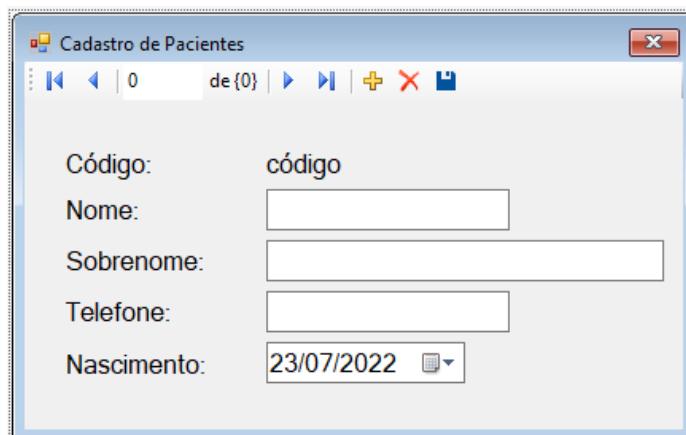
Vamos configurar nossa fonte de dados. Para isso, deixe o formulário de pacientes em modo de Design e abra a janela Fontes de Dados (SHIFT + ALT + D). Clique no botão ao lado de Paciente e troque o formato dele para Details, como mostra imagem à direita:



Expanda a tabela Paciente e troque o formato do IDPaciente para Label, da mesma forma como fizemos no formulário anterior. Após isso, arraste a tabela Paciente para o form recém-criado:



Altere algumas propriedades de seu form para que o mesmo fique padronizado como os outros, parecido com o da imagem abaixo:



Agora vamos ao form principal e codificar o evento click do segundo botão, para fazer a chamada a esse form que criamos. No form principal, clique duas vezes no botão Pacientes e codifique o trecho abaixo:

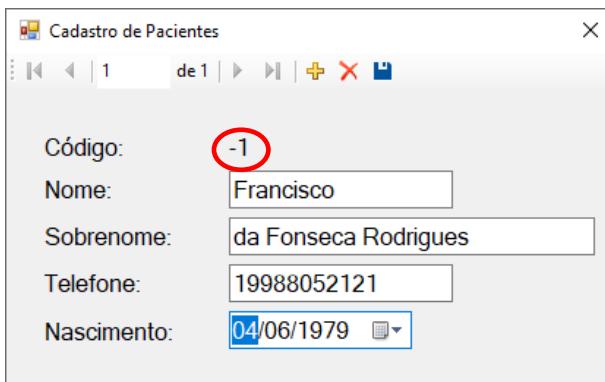
```
private void btnPacientes_Click(object sender, EventArgs e)
{
    frmPacientes paciente = new frmPacientes();
    paciente.ShowDialog();
}
```

Fizemos acima o mesmo que foi feito no botão do Cadastro de Médicos: instanciamos o formulário de pacientes e o chamamos para exibição com o método `ShowDialog()`, no evento `Click` do botão.

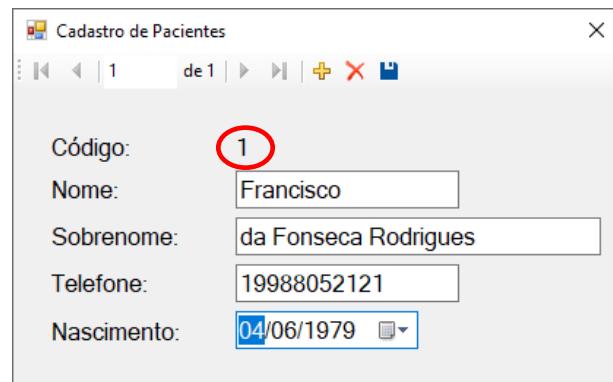
Abra o formulário de Pacientes em modo de código (tecla F7 no formulário) e, no evento `frmPacientes_Load`, digite o código abaixo, para fornecer sua senha à string de conexão ao servidor:

```
private void frmPacientes_Load(object sender, EventArgs e)
{
    pacienteTableAdapter.Connection.ConnectionString += ";Password=suasenha";
    // TODO: esta linha de código carrega dados na tabela 'chicoDataSet.Paciente'.
    // Você pode movê-la ou removê-la conforme necessário.
    this.pacienteTableAdapter.Fill(this.chicoDataSet.Paciente);
}
```

Experimente adicionar alguns registros para testar as funcionalidades que o Visual Studio nos fornece com apenas alguns cliques. Como se pode ver, não é nada muito complexo de ser feito.



Após pressionar e digitar os dados



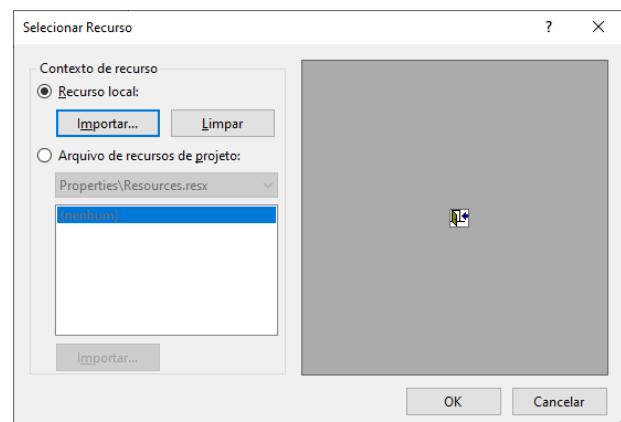
Após pressionar

Agora no form de Cadastro de Médicos, adicione um separador (Separator) e mais um botão, por meio das opções do `BindingNavigator`. Clique com o botão direito nesse novo botão, e selecione em `DisplayStyle > Image`. Agora clique neste botão, abra as propriedades do seu form, mude seu nome para `btnFechar`. Na propriedade `Image`, clique no botão com reticências [...] e, após ter sido aberta a janela **Selecionar Recurso**, pressione o botão `[Importar]` e importe a imagem `Close1.bmp` fornecida pelo professor, e digite `Fechar`, como mostra a imagem ao lado:

Clique duas vezes nesse botão e codifique à chamada ao método `Close()` do formulário.

```
private void btnFechar_Click(object sender, EventArgs e)
{
    Close();
}
```

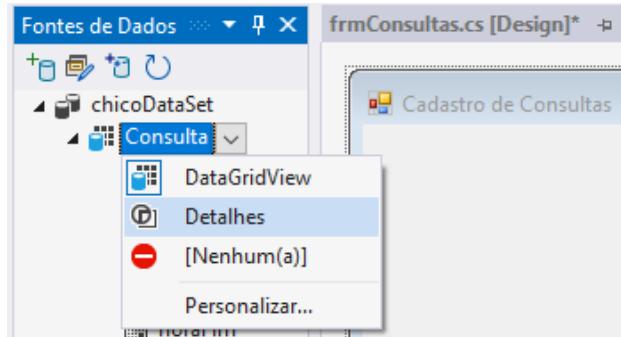
Faça isso aos demais formulários, mantendo aquela ideia de criar uma padronização aos forms.



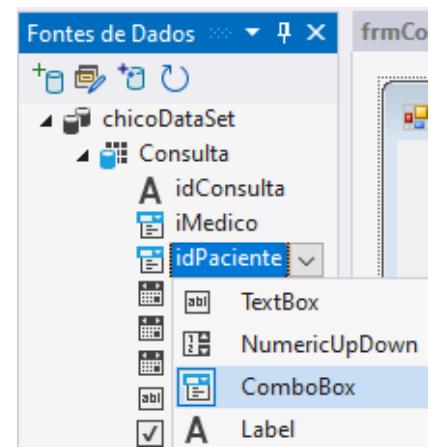
Cadastro de Consultas

Crie um novo form e dê o nome de frmConsultas. Aplique as configurações padronizadas que definimos no início do artigo, clique no evento KeyDown do form e insira novamente o código para que o form se feche quando o usuário teclar ESC.

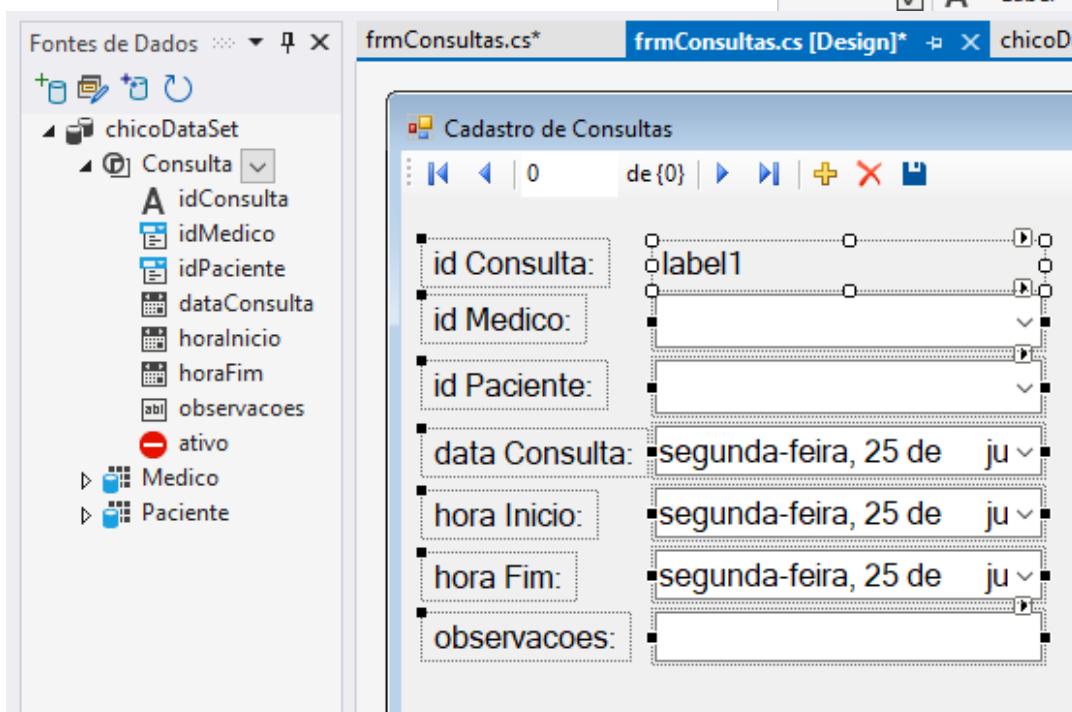
Abra a janela **Fontes de Dados** (SHIFT+ALT+D), clique no botão ao lado de Consulta e troque o formato dele para **Detalhes**, como fizemos anteriormente.



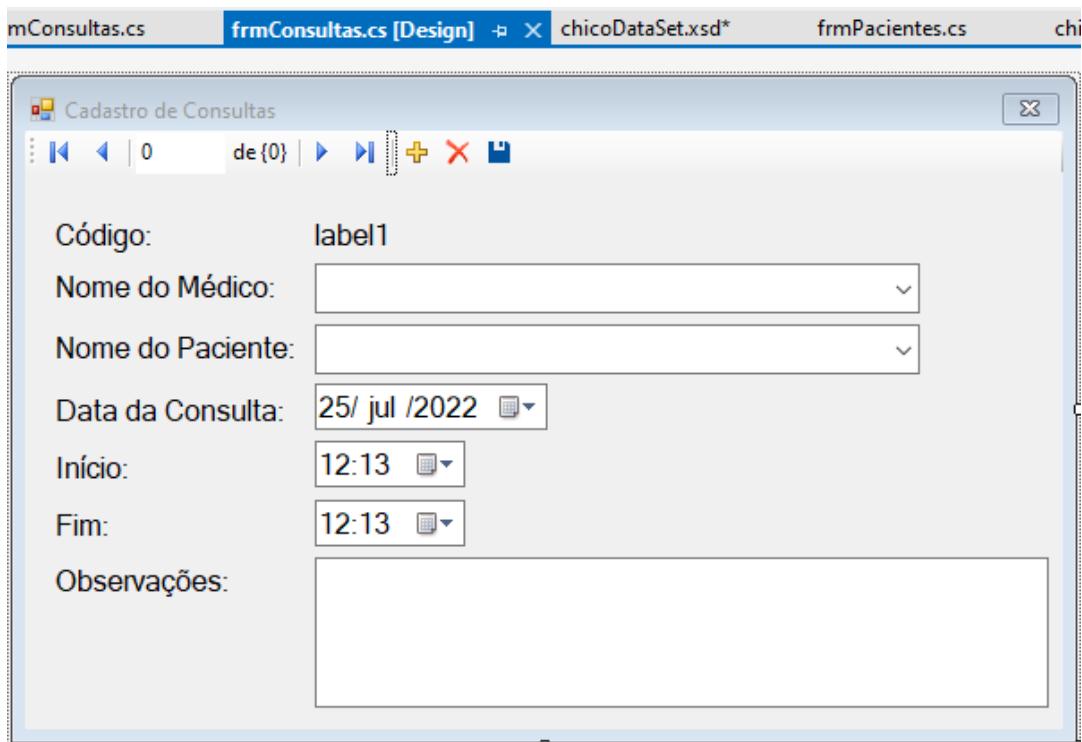
Agora expanda a tabela Consulta e troque o formato do IDConsulta para Label, idMedico troque para um ComboBox para que o usuário possa escolher qual médico será mostrado no momento da consulta e faça o mesmo para a coluna IDPaciente. E na coluna Ativo, deixe como Nenhum(a), porque não precisaremos mostrar esta coluna ao usuário, ela servirá para controle interno, para sabermos se tal consulta está ativa ou não.



Após isso, arraste a tabela Consulta para o form recém-criado:



Altere as propriedades do form para que fiquem semelhantes às da imagem abaixo:



Nos comboboxes Nome do Médico e Nome do Paciente, altere a propriedade **DropDownStyle** para DropDownList, para que o usuário não possa digitar nos combos. No combo Data, altere a propriedade **Format** para Custom e a propriedade **CustomFormat** para dd/MMM/yyyy. Nos combos Início e Término, altere a propriedade **Format** para Custom e a propriedade **CustomFormat** para HH:mm. Assim você coloca um valor personalizado tanto para data quanto para hora. No textboxObservacoes, altere a propriedade **Multiline** para True, para deixá-lo com várias linhas para inserir as observações. Ancore o textboxObservacoes dos lados direito e fundo, também. As demais configurações seguem o padrão dos outros forms.

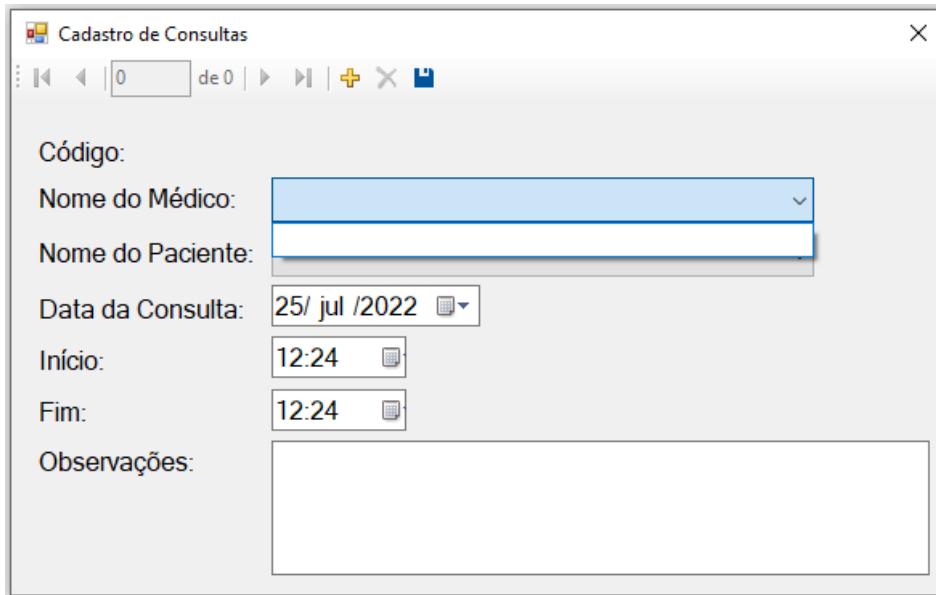
No evento Load desse formulário, inclua a linha de informação da senha de acesso ao banco de dados, como vemos no código abaixo:

```
private void frmConsultas_Load(object sender, EventArgs e)
{
    consultaTableAdapter.Connection.ConnectionString += ";Password=suaSenha";
    //TODO:esta linha de código carrega dados na tabela 'chicoDataSet.Consulta'.
    //Você pode movê-la ou removê-la conforme necessário.
    this.consultaTableAdapter.Fill(this.chicoDataSet.Consulta);
}
```

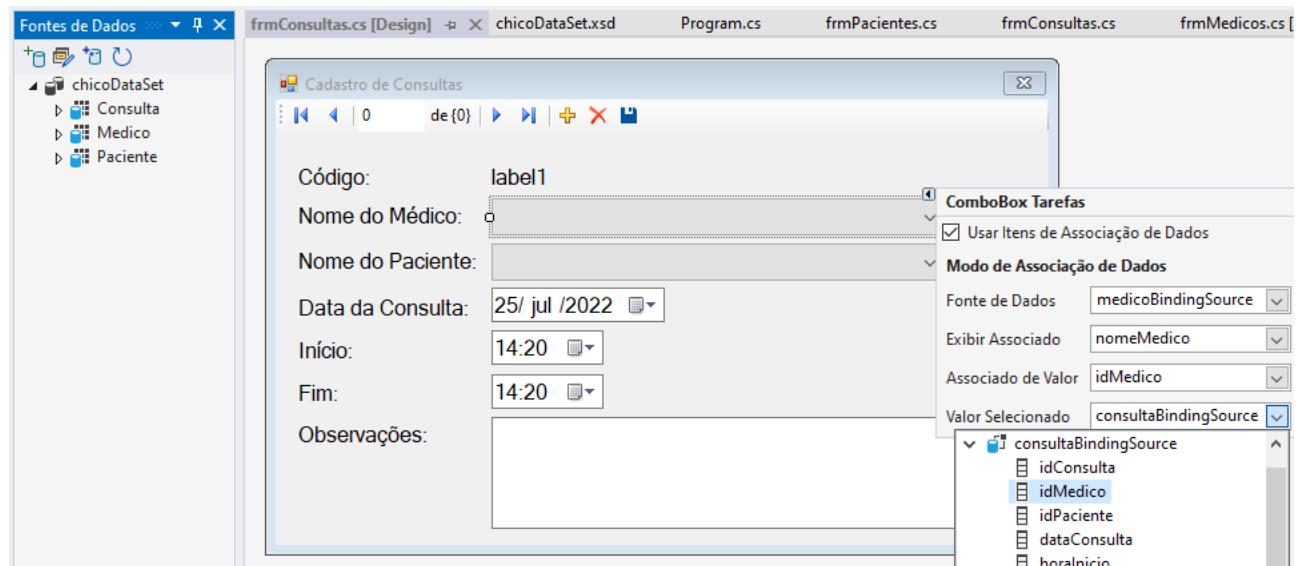
Agora volte ao form principal, adicione um novo botão, dê dois cliques nele e faça a chamada ao formulário de Consulta por meio do código seguinte:

```
private void btnConsultas_Click(object sender, EventArgs e)
{
    frmConsultas consulta = new frmConsultas();
    consulta.ShowDialog();
}
```

Agora, se compilarmos nosso sistema, veremos que o form de Consulta não nos retorna absolutamente nada nos combos de Médico e Paciente.



Isso acontece porque não relacionamos as respectivas tabelas aos combos. Para fazer isso, abra a janela **Fonte de Dados** (SHIFT + ALT + D) e arraste a tabela Médicos para o seu respectivo combobox. Faça o mesmo com a tabela Paciente. Após isso, podemos clicar ao lado do combo, na setinha e percebermos que ele arrasta os dados referentes ao médico para este combo:



Essa operação é chamada de **lookup**, e consiste em buscar dados de uma tabela a partir de um dado de outra tabela. Acima, os nomes de médicos e pacientes estarão associados às respectivas chaves primárias, idMedico e idPaciente. Essa operação fez com que as tabelas Medico e Paciente também fossem incorporadas a esse formulário, de forma que precisaremos também informar, para cada uma delas, a senha de acesso ao servidor, no evento Load, como vemos abaixo:

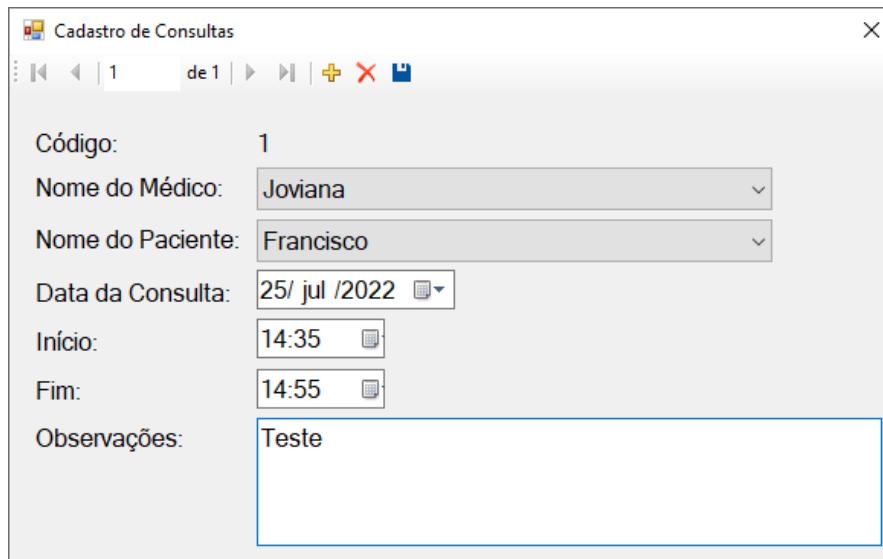
```
private void frmConsultas_Load(object sender, EventArgs e)
{
    pacienteTableAdapter.Connection.ConnectionString += ";Password=suaSenha";
    medicoTableAdapter.Connection.ConnectionString += ";Password=suaSenha";
    consultaTableAdapter.Connection.ConnectionString += ";Password=suaSenha";
    //TODO: esta linha de código carrega dados na tabela 'chicoDataSet.Paciente'.
    //Você pode movê-la ou removê-la conforme necessário.
    this.pacienteTableAdapter.Fill(this.chicoDataSet.Paciente);
    ...
}
```

```

        this.medicoTableAdapter.Fill(this.chicoDataSet.Medico);
        ...
        this.consultaTableAdapter.Fill(this.chicoDataSet.Consulta);
    }
}

```

Agora execute o sistema e tente adicionar uma nova consulta.



Como você deve ter notado, apenas o prenome do médico e do paciente aparecem. Isso era esperado, pois separamos o nome do sobrenome (como advogada a Primeira Forma Normal) e a ferramenta do Visual Studio só nos permite exibir um campo da tabela (Exibir Associado) do lookup.

Na tabela de Médicos e na tabela de Pacientes, precisaremos incluir um campo adicional que seja resultante da concatenação entre prenome e sobrenome. Esse campo deverá ser “calculado” para cada registro de médico e de paciente. Vamos chamar esse campo de nomeCompleto. Esse campo deve ser incorporado ao comando Select que traz os registros dessas tabelas.

Assim, entre novamente na guia do DataSet e clique no Adapter da tabela Medico (MedicoTableAdapter). Na propriedade SelectCommand, clique no botão [...] e abra o Construtor de Consultas. Complemente o comando SELECT após **crm** incluindo a expressão “**, nomeMedico+ ‘ ‘ + sobrenomeMed as nomeCompleto**”, e pressione [Executar Consulta]:

Coluna	Alias	Tabela	Saída	Tipo de Class...	Ordem d
idMedico		Medico (C...)	<input checked="" type="checkbox"/>		
nomeMedico		Medico (C...)	<input checked="" type="checkbox"/>		
sobrenomeMed		Medico (C...)	<input checked="" type="checkbox"/>		
crm		Medico (C...)	<input checked="" type="checkbox"/>		
nomeMedico + ' ' + sobrenomeMed	nomeCompleto		<input checked="" type="checkbox"/>		


```

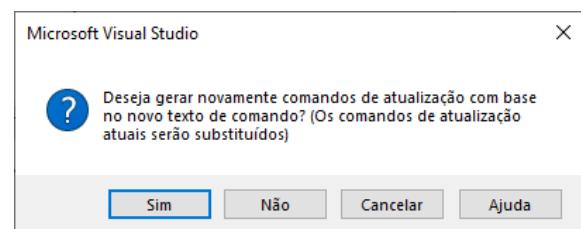
SELECT idMedico, nomeMedico, sobrenomeMed, crm, nomeMedico + ' ' + sobrenomeMed AS nomeCompleto
FROM Cons.Medico

```


	idMedico	nomeMedico	sobrenomeMed	crm	nomeCompleto
►	1	Florencio	Gaudencio	1212	Florencio Gaudencio
	2	Joviana	Silva Souza	3343	Joviana Silva Souza
*	3	Verônica	Chaves	18721	Verônica Chaves
	NULL	NULL	NULL	NULL	NULL

A célula é Somente Leitura.

Será perguntando se você deseja atualizar os comandos, responda que sim:



Agora, você deve seguir os mesmos passos para complementar o comando Select da tabela Paciente, adicionado a ele o campo nomeCompleto, como vemos abaixo. Também responda Sim para a pergunta sobre atualização dos comandos:

Por fim, atualize os comboboxes do formulário de Consultas, para que o campo nomeCompleto seja usado para exibição (Exibir Associado), tanto para Médicos quanto para Pacientes:

Execute o sistema, adicione mais algumas consultas e observe que, agora, aparecem os nomes completos dos médicos e dos pacientes, o que facilitará bastante para o usuário cadastrar consultas corretamente.

Esse é um aplicativo simples, com poucos recursos e ainda precisará ser melhorado.

Mas, antes disso, aprenderemos técnicas e abordagens que tornarão nosso banco de dados mais eficiente e mais adequado para acesso programado.

6. Otimização de consultas: Check, Views e Index

6.1. Check

Check ou **Restrição de Verificação** é uma cláusula usada na definição de campos de tabelas que permite estabelecer condições de verificação da consistência de valores inseridos ou alterados nesses campos. Ela é usada para especificar quais valores são aceitáveis em um ou mais campos. Como se trata de uma condição, usamos expressões lógicas que retornem true ou false para especificar o critério de validação do campo.

Dentro da criação de uma tabela, Check envolve comparações como vemos no exemplo a seguir:

```
create table Estoque.Venda
(
    idVenda int identity primary key,
    idProduto int not null,
    dataInclusaoNoEstoque date not null,
    quantidadeVendida float not null check (quantidadeVendida >= 0),
    CPFComprador char(14)
        CHECK (CPFComprador like '[0-9][0-9][0-9].[0-9][0-9][0-9].[0-9][0-9][0-9]-[0-9][0-9]'),
    CEPComprador char(8) CHECK (CEPComprador like '[0-9][0-9][0-9][0-9][0-9]-[0-9][0-9]'),
    dataVenda Date CHECK (dataVenda >= dataInclusaoNoEstoque),
    foreign key (idProduto) references Estoque.Produto(idProduto)
)
```

6.2. Visões

Uma View (visão, visualização) na terminologia SQL é uma única tabela que é derivada de outras tabelas. Essas outras tabelas podem ser tabelas da base de dados ou outras visões previamente definidas. Uma visão não necessariamente existe na forma física; é considerada como uma tabela **virtual**, ao contrário das tabelas da base de dados, cujas tuplas são sempre armazenadas fisicamente no banco de dados. Isto limita possíveis operações de atualização que podem ser aplicadas às visões, mas não fornece nenhuma limitação na consulta de uma visão.

Uma visão contém linhas (registros) e colunas (campos) como uma tabela real, e pode receber cláusulas como JOIN, WHERE e funções como se fosse uma tabela real. A diferença é que o resultado não fica armazenado no banco de dados, e é construído a cada momento em que a visão é invocada (mediante uma query), além de poder envolver diversas tabelas “físicas” relacionadas.

Podemos pensar em uma visão como uma forma de deixar pronta uma consulta a uma ou mais tabelas que precisamos acessar frequentemente, mesmo que possa não existir fisicamente.

Por exemplo, referindo-se ao [Banco de Dados da Empresa](#), podemos ter de, frequentemente, emitir consultas que recuperam o nome do funcionário e os nomes dos projetos em que o funcionário trabalha. Ao invés de ter que especificar a junção das três tabelas EMPREGADO, TRABALHA_EM e PROJETO a cada vez que emitimos esta consulta, podemos definir uma visão que é especificada como o resultado de estas junções. Em seguida, podemos emitir consultas sobre a visão, que são especificadas como um select em uma única tabela virtual, ao invés de selects envolvendo duas junções em três tabelas.

Chamamos as tabelas EMPREGADO, TRABALHA_EM e PROJETO de tabelas definidoras.

O programa de aplicação consulta a visão já pronta, ao invés de ter de especificar um comando select mais complexo. A consulta à visão mostrará sempre resultados atualizados, pois o servidor de banco de dados sempre recaria os resultados toda vez que uma aplicação consulta essa visão.

Outra vantagem de usar visões é que podemos especificar quais usuários podem ou não acessar essa visão, de acordo com suas prerrogativas de acesso. A especificação de controle de privilégios de acesso é feita com os comandos GRANT e REVOKE.

Por exemplo, em uma escola, o Setor de Orientação Educacional pode ter acesso a todos os dados dos alunos, enquanto que um professor poderá ter acesso a apenas uma parte desses dados. Podemos, portanto, criar uma visão para o Setor de Orientação Educacional que envolva todos os dados (tabelas, relacionamentos, etc.) e dar acesso (GRANT) aos usuários desse setor e negamos (REVOKE) acesso aos professores, enquanto criamos uma visão mais limitada para os professores e a ela damos acesso aos professores e também aos usuários do Setor de Orientação.

6.2.1. Especificação das Visões em SQL e seu uso

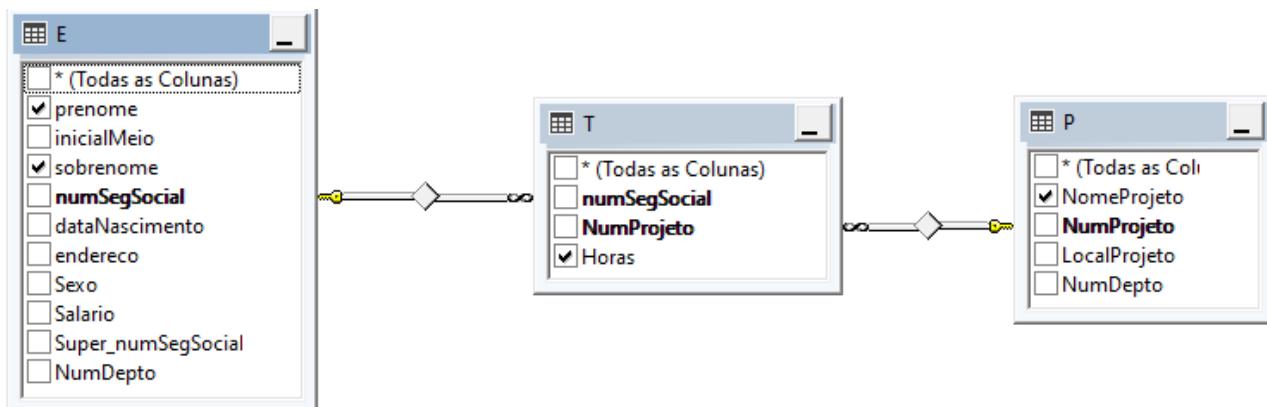
Em SQL, o comando para especificar uma visão é CREATE VIEW.

Nele, é dado à visão um nome de visão (ou nome de tabela virtual), uma lista de nomes de atributos e uma consulta para especificar o conteúdo da visão. Se nenhum dos atributos da visão resultar da aplicação de funções ou operações aritméticas, não temos que especificar novos nomes de atributos para a visão, já que seriam os mesmos que os nomes dos atributos da definição das tabelas no caso padrão.

As visões em V1 e V2 criam tabelas virtuais cujos diagramas são ilustrados na figura abaixo, quando aplicados ao esquema do [Banco de Dados da Empresa](#).

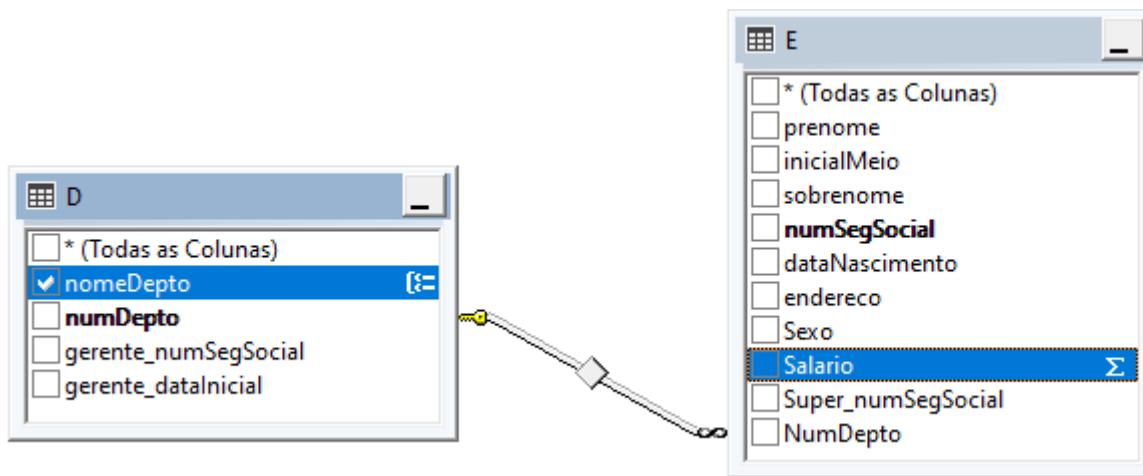
Visão V1:

```
CREATE VIEW Empresa.V_Onde_Trabalha
AS
  SELECT
    E.prenome, E.sobrenome, P.NomeProjeto, T.Horas
  FROM
    (Empresa.Empregado E JOIN Empresa.TRABALHA_EM T
      ON E.numSegSocial = T.numSegSocial)
    JOIN Empresa.PROJETO P ON T.NumProjeto = P.NumProjeto;
```



Visão V2:

```
CREATE VIEW Empresa.Info_Depto
AS
  SELECT
    nomeDept Departamento, COUNT(*) '#Empregados',
    SUM(Salario) 'Salário total'
  FROM
    Empresa.Departamento D JOIN Empresa.Empregado E
      ON D.numDept = E.NumDept
  GROUP BY
    nomeDept;
```



Agora podemos especificar consultas SQL em uma visão - ou tabela virtual - da mesma forma que nós especificamos consultas envolvendo tabelas do banco de dados. Por exemplo, para recuperar todos os dados dos empregados alocados a projetos, usamos o comando Select abaixo:

Consulta V1.1:

```
Select * from empresa.V_Onde_Trabalha
```

O resultado está ao lado. Note que a complexidade do **Select** que deu origem ao código da visão está oculta na própria visão.

Uma aplicação, portanto, poderia usar apenas o comando anterior que consulta a visão, sem ter de se preocupar com criar o comando que faz a junção entre as três tabelas.

	prenome	sobrenome	NomeProjeto	Horas
1	John	Smith	ProdutoX	32.5
2	John	Smith	ProdutoY	7.5
3	Franklin	Wong	ProdutoY	10.0
4	Franklin	Wong	ProdutoZ	10.0
5	Franklin	Wong	Informatização	10.0
6	Franklin	Wong	Reorganização	10.0
7	Joyce	English	ProdutoX	20.0
8	Joyce	English	ProdutoY	20.0
9	Ramesh	Narayan	ProdutoZ	40.0
10	James	Borg	Reorganização	40.0
11	Jennifer	Wallace	Reorganização	15.0
12	Jennifer	Wallace	Novos Benefíc	20.0
13	Ahmad	Jabbar	Informatização	35.0
14	Ahmad	Jabbar	Novos Benefíc	5.0
15	Alicia	Zelaya	Informatização	10.0
16	Alicia	Zelaya	Novos Benefíc	30.0

Se quiséssemos consultar apenas os nomes completos de todos os funcionários que trabalham no projeto 'ProductX', podemos utilizar a visão Empresa.V_Onde_Trabalha e especificar a consulta como abaixo, com o resultado à direita:

Consulta V1.2:

```
SELECT
    prenome, sobrenome
from
    Empresa.V_Onde_Trabalha
where
    nomeProjeto = 'ProdutoX';
```

	prenome	sobrenome
1	John	Smith
2	Joyce	English

A mesma consulta exigiria a especificação de duas junções se especificada na base de dados diretamente; uma das principais vantagens de uma visão é simplificar a especificação de certas consultas. As visões também são utilizadas como mecanismo de segurança e autorização, como comentamos anteriormente.

Uma visão sempre estará atualizada; se modificarmos os registros nas tabelas originais sobre as quais a visão é definida, a visão deverá refletir automaticamente estas mudanças quando for consultada novamente. Portanto, a visão não tem que ser realizada ou materializada no momento de sua definição, mas sim no momento em que especificamos uma consulta na mesma. É responsabilidade do SGBD e não do usuário para garantir que a visão seja mantida atualizada.

Abaixo temos a consulta aos resultados da visão Info_Depto:

Consulta V2.1:

```
select
    *
from
    empresa.info_dept
order by
    'Salário total' desc
```

Quando você executar os comandos Create View no servidor de banco de dados SQL Server, estará criando duas visões no item Exibições de seu banco de dados, como vemos ao lado:

Visões não podem usar a cláusula Order By a menos que exista uma cláusula TOP na lista de seleção do SELECT. Também não podem referenciar tabelas temporárias nem usar a palavra reservada INTO.

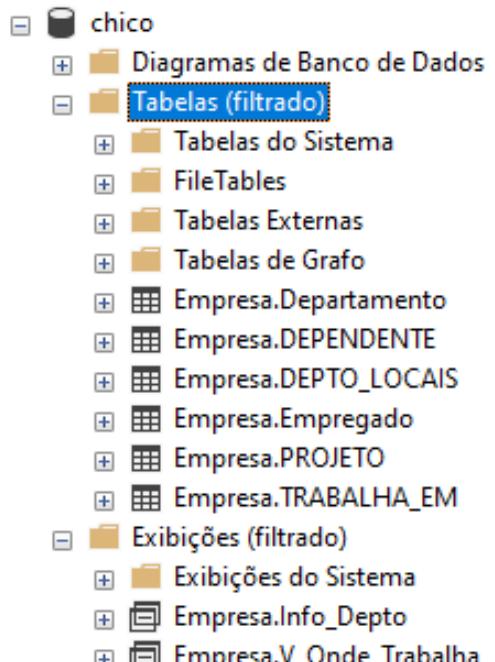
É possível modificar os dados de uma tabela base subjacente através de uma visão, contanto que as seguintes condições sejam verdadeiras:

- Todas as modificações, inclusive as instruções UPDATE, INSERT e DELETE, devem referenciar **campos de apenas uma tabela base**.
- Os campos a serem modificados na visão devem referenciar diretamente os dados subjacentes das colunas da tabela. As colunas não podem ser derivadas de qualquer outro modo como, por exemplo:
 - Uma função de agregação: AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR e VARP.
 - Campo calculado: o campo não pode ser calculado a partir de uma expressão que utiliza outros campos. As colunas formadas com o uso dos operadores de conjunto UNION, UNION ALL, CROSSJOIN, EXCEPT e INTERSECT resultam em um campo calculado e também não são atualizáveis.
- Os campos modificados não são afetados pelas cláusulas GROUP BY, HAVING ou DISTINCT.
- TOP não é usado em nenhum lugar no SELECT da visão junto com a cláusula WITH CHECK OPTION.

As restrições anteriores aplicam-se a todas as subconsultas da cláusula FROM da visão, exatamente como se aplicam à própria visão. Em geral, o Mecanismo de Banco de Dados deve ser capaz de rastrear sem ambiguidade as modificações da definição da exibição em uma tabela base. Para obter mais informações, confira [Modificar dados por meio de uma visão](#).

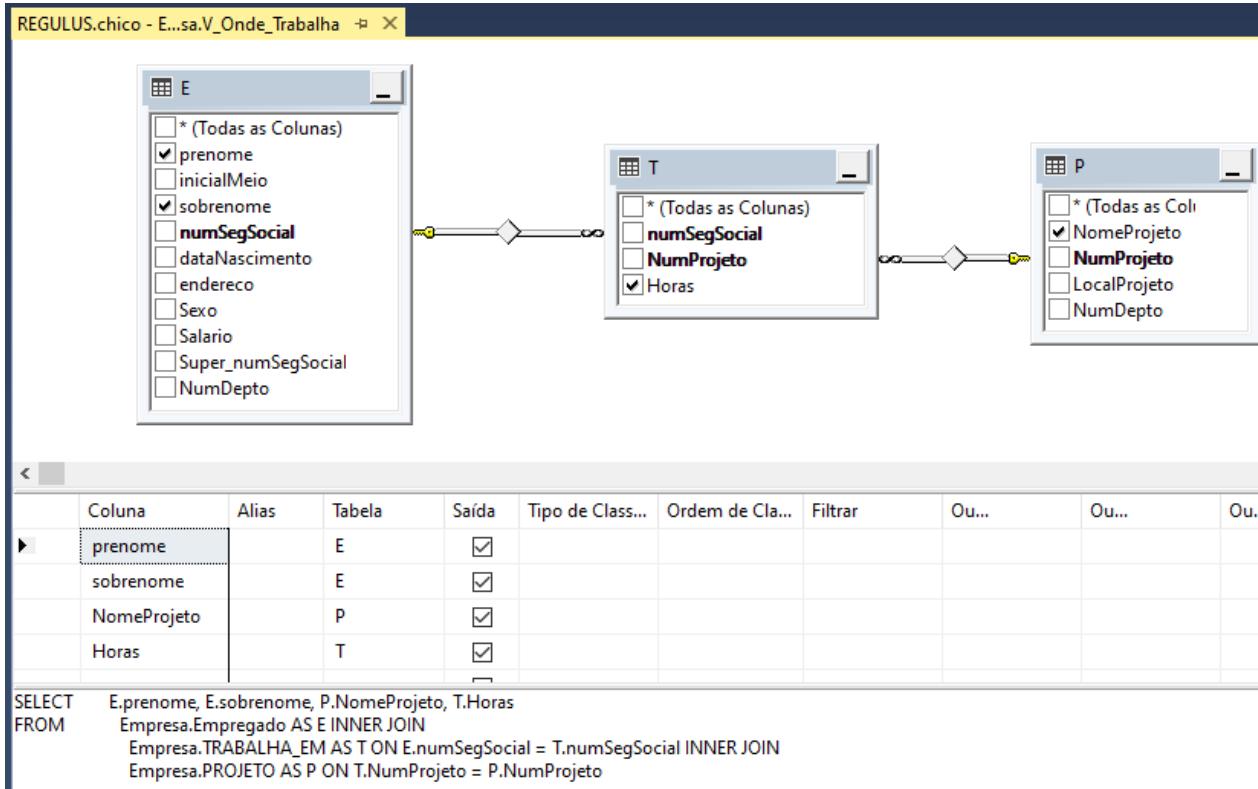
Você pode usar a ferramenta visual de design de visões do Sql server. Basta clicar com o botão direito no item Exibições do Pesquisador de Objetos do seu banco de dados e clicar em [Nova Exibição...]. Você poderá adicionar as tabelas desejadas, bem como visões já criadas anteriormente (por comandos ou pelo designer visual). Isso causará a abertura da janela de design de visões, na qual você poderá adicionar tabelas, relacioná-las, selecionar campos que deseja

	Departamento	#Empregados	Salário total
1	Pesquisa	4	133000.00
2	Administração	4	121000.00
3	Sede	1	55000.00
4	Tecn Informação	2	50000.00



serem mostrados no resultado, dentre outras ações, que podem ser feitas visualmente ao invés de codificadas.

Você pode clicar com o botão direito em uma visão já existente e selecionar a opção [Design] do menu suspenso que aparecerá, para editá-la na ferramenta visual.



Se não precisarmos mais de uma visão, podemos usar o comando **DROP VIEW** para dispor dela. Por exemplo, para nos livrarmos da visão V1, podemos usar a instrução SQL abaixo:

```
DROP VIEW Empresa.V_Onde_Trabalha
```

É bastante importante criar visões para as principais operações de consulta ao seu banco de dados. Assim, comandos de certas consultas já ficam pré-definidas no servidor e o programador de aplicações não precisa se debruçar sobre a complexidade dessas consultas, de forma a tornar o processo de desenvolvimento de aplicações menos sujeito a erros e a atrasos.

Também é possível configurar permissões de acesso a visões de acordo com os perfis de cada usuário de seus sistemas de banco de dados, permitindo um ajuste fino de quem pode ou não acessar determinadas informações. Assim, ao criar visões apropriadas e permitir acesso a elas para certos usuários e não às tabelas base, tais usuários ficam restritos a ver apenas os dados recuperados pela visão, não tendo acesso aos dados completos das tabelas base.

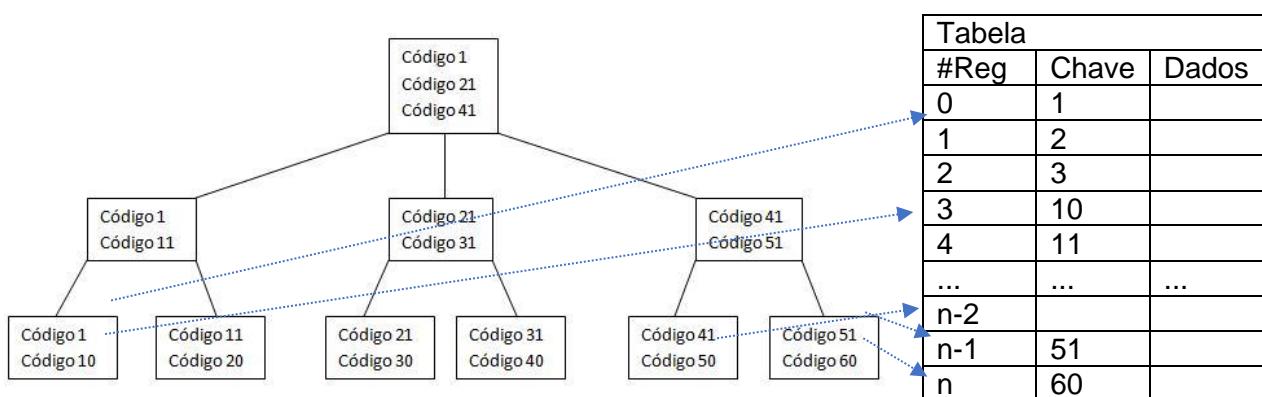
EXERCÍCIOS

1. Usando o esquema Cons, use o comando **Create View** para criar uma visão que liste todas as consultas ativas dos médicos. Deve-se mostrar os nomes completos do médico e do paciente, a data e a hora de início das consultas. Essa visão deve se chamar **V_Consultas_Ativas_Medicos**.
2. Usando o esquema Cons, use a ferramenta visual de criação de visão para criar uma visão que liste os pacientes e suas consultas, indicando os nomes do médico e do paciente, a data e a hora de início e fim das consultas. Essa visão deve se chamar **V_Consultas_Pacientes**. Para ativar essa ferramenta, clique com o botão direito no item **Exibições** de seu banco de dados e selecione a opção **[Criar Exibição...]**. Você pode usar uma visão já existente para criar outra visão, além das tabelas do banco de dados.

6.3. Índices

Um índice é uma estrutura adicional às tabelas e visões de um banco de dados que permite aumentar o desempenho nos acessos, buscando diminuir a ocorrência de leituras sequenciais e tornando a busca mais rápida e eficiente, de forma semelhante ao que se faz em pesquisa binária em um vetor.

Você cria um índice através do comando Create Index, e associa essa nova estrutura a uma tabela, usando um grupo de campos dessa tabela para compor uma chave de pesquisa. O Sql Server utiliza, internamente, uma estrutura de dados chamada B-Tree (Árvore B) para armazenar ligações entre as chaves de pesquisa e os registros a que elas se referem. Abaixo temos uma ilustração de uma possível B-Tree para uma chave de pesquisa associada a um campo inteiro qualquer de uma tabela:



Uma busca pelo índice inicia-se no nível superior (chamado de raiz) percorrendo todas as linhas até achar a cadeia de valores a qual o mesmo se encaixa e através da ligação (um ponteiro) pular para a página do nível intermediário que o mesmo se refere. No nível intermediário repete o mesmo processo até achar a cadeia de valores e pular para a página de nível inferior (chamado de folha) conforme a ligação. No nível folha novamente repete-se o processo até achar o valor desejado e nesse momento são localizados os dados desejados (ou descobre-se que não existem).

Por exemplo, conforme a **figura acima**, para achar o código 23 iniciaria a busca pelo nível raiz percorrendo as linhas. Como o código 23 está entre 21 e 41 o SQL Server calcula que o código 23 se encontra na sequência do código 21 e pula para a página do nível intermediário que contém os valores 21 a 31. Em seguida analisaria que a primeira opção (21) se encaixa para a busca e pularia para a página de nível folha que contém a cadeia de 21 a 30, percorreria a mesma até achar o código 23 e finalizaria a busca.

A cada escolha de um caminho nos ramos da árvore B, deixamos de percorrer vários outros caminhos (onde a chave procurada não teria como estar armazenada) e, assim, a pesquisa acaba por se tornar bastante rápida.

Pode parecer, portanto, que deveríamos criar índices para todas as consultas que precisarmos fazer, de forma que elas fiquem extremamente rápidas. No entanto, a existência de um índice torna as operações INSERT, DELETE, UPDATE e MERGE mais lentas. Cada vez que houver uma inclusão, alteração ou exclusão, o SQL Server precisa atualizar a tabela e os índices a ela associados. Obviamente isso toma mais tempo do que se não tivéssemos índices. Ou seja, usar índices é importante, mas com parcimônia, para que apenas as operações mais críticas em termos de velocidade de consulta sejam indexadas.

O comando Create Index é descrito abaixo:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WITH ( [ ,...n ] ) ]
```

```
[ ON { partition_scheme_name ( column_name )
      | filegroup_name
      | default
    }
]
[ ; ]
```

O comando abaixo criará um índice, nomeado ixEmpregadoDept, associado à tabela Empregado e que usa como chave de pesquisa o campo numDept. Dessa forma, quando fizermos consultas nessa tabela que envolvam o campo numDept, as pesquisas ficarão mais rápidas:

```
Create index ixEmpregadoDept
on Empresa.Empregado (numDept)
```

Mas não criamos índices para salários ou sexo, que são valores para os quais não teremos necessidade de muitas consultas.

Você pode usar vários campos na expressão de indexação ON, sendo que o desempate será da esquerda para a direita. Por exemplo:

```
Create index ixProjetoDept
on Empresa.Projeto (numDept, numProjeto)
```

Para remover um índice, usamos o comando Drop Index, o nome do índice e da tabela à qual ele está associado:

```
drop index ixEmpregadoDept on Empresa.Empregado
```

Dicas a serem consideradas ao criar índices:

Campos para serem indexados par aumentar o desempenho em consultas:

- Chaves Primárias;
- Chaves Estrangeiras;
- Colunas acessadas por intervalos (between);
- Campos utilizados em group by ou order by;

Campos que não devem ser indexados:

- Campos dos tipos: text, image, decimais;
- Campos calculados;
- Campos com alta cardinalidade (Masculino ou Feminino);

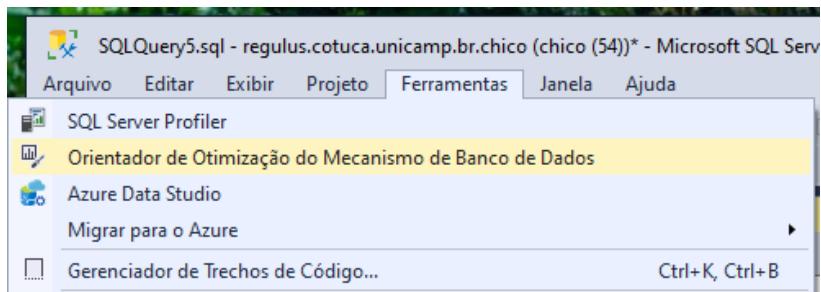
Use muitos índices para aperfeiçoar o desempenho da consulta em tabelas com baixos requisitos de atualização, mas com grandes volumes de dados. Grandes números de índices podem ajudar o desempenho de consultas que não modificam dados, como instruções SELECT, porque o otimizador de consulta tem mais índices para escolher para determinar o método de acesso mais rápido.

Indexar tabelas pequenas pode não ser bom porque pode fazer o otimizador de consulta levar mais tempo para atravessar o índice procurando dados do que executar uma simples varredura de tabela.

Portanto, os índices em tabelas pequenas talvez nunca sejam usados, mas ainda devem ser mantidos como dados nas alterações de tabela.

Índices em visões podem prover ganhos de desempenho significativos quando a visão contiver agregações, junções de tabela ou uma combinação de agregações e junções. A visão não precisa estar explicitamente referenciada na consulta para o **otimizador de consulta** usá-la.

O otimizador de consultas é um processo do Servidor SQL Server que analisa cada consulta efetuada num banco de dados e procura sugerir mudanças para torná-las mais eficientes.



Tipos de Índices

O SQL Server possui vários tipos de índices, cada um adequado a um tipo de situação. Vejamos um pouco sobre eles:

Índice Cluster

Uma tabela pode ter somente **um índice cluster**. Essa limitação existe porque este tipo de índice classifica e armazena as linhas da tabela com base nos valores da coluna escolhida para fazer parte do índice, e as linhas só podem ser classificadas fisicamente em uma única ordem. Se você escolheu o código de usuário para ser o índice cluster da sua tabela de usuário, então todos os registros da sua tabela estarão ordenados fisicamente pelo código do usuário.

É importante você saber que um índice cluster pode ter uma ou várias colunas.

A sintaxe para criar um índice é muito simples... Você precisa “dizer” para o SQL Server o seguinte: Crie um índice **cluster** chamado Ix_Responsavel na tabela DEPENDENTE e que contenha a coluna NumSegSocial. Com isso, o servidor procurará agrupar os dependentes em grupos definidos pelo campo numSegSocial.

Traduzindo para o Transact-SQL ficaria assim:

```
CREATE CLUSTERED INDEX Ix_Responsavel
    ON Empresa.Dependente (NumSegSocial);
```

Mas, ao executarmos esse comando, notaremos que haverá erro, pois essa tabela já possui um índice clusterizado, formado a partir da chave primária:

```
Mensagem 1902, Nível 16, Estado 3, Linha 6
Não é possível criar mais de um índice clusterizado na tabela tabela
'Empresa.Dependente'.
Cancele o índice clusterizado 'PK__DEPENDEN__750150137648FEED' existente antes
de criar outro.
```

Horário de conclusão: 2022-10-13T22:03:12.4885186-03:00

Assim, como é recomendado que toda tabela possua uma chave primária, o SQL Server já usa essa chave primária para compor um índice clusterizado padrão.

Você pode, se desejar, remover esse índice padrão e criar o seu próprio, como no comando acima. Isso não fará com que a tabela deixe de possuir a chave primária.

Índice Não-Clusterizado

Um índice não-cluster é uma estrutura de índice **separada** dos dados armazenados em uma tabela. Eles costumam ser uma forma mais rápida de localizar dados do que a busca na tabela sem índice. Geralmente, os índices não-cluster são criados para aprimorar o desempenho de consultas realizadas com frequência, e que contém colunas diferentes daquelas que fazem parte do índice cluster. Você pode criar vários índices não cluster em uma tabela.

Para criar um índice cluster temos que dizer para o SQL Server: Crie um índice **não-cluster** chamado ix_Responsavel na tabela DEPENDENTE e que contenha a coluna NumSegSocial.

```
CREATE NONCLUSTERED INDEX Ix_Responsavel
    ON Empresa.Dependente (NumSegSocial);
```

Como se pode ver, o índice acima substitui com êxito o que tentamos fazer anteriormente e não pudemos, por já haver um índice clusterizado associado a essa tabela. Também podemos criar mais índices nessa mesma tabela usando, por exemplo, o campo Relacionamento. Assim, se tivermos necessidade de consultas que envolvam esse campo e sejam bastante frequentes, valeria a pena criar o índice abaixo:

```
CREATE NONCLUSTERED INDEX Ix_Depend_Relacionamento
    ON Empresa.Dependente (Relacionamento);
```

O tipo de índice Não-clusterizado pode ter duas particularidades interessantes:

- Índice **filtrado** – Determina que só os valores que obedecem a uma determinada condição devem ser indexados. Por exemplo, na nossa tabela de dependentes, queremos filtrar somente os registros de relacionamento 'filho' ou 'filha':

```
CREATE NONCLUSTERED INDEX Ix_Depend_Filhos
    ON Empresa.Dependente (Relacionamento)
        Where relacionamento in ('filho', 'filha'); -- não aceitou LIKE 'filh%'
```

- Índices com **colunas incluídas** - Imagine que você está procurando um responsável por um relacionamento, através do NumSegSocial. No índice ix_Responsavel a informação principal que você vai usar para achar os registros que lhe interessam, mas associados a cada NumSegSocial existem outros dados e você gostaria de ter acesso rápido a eles também. Ou seja, o índice é feito pelo NumSegSocial, mas junto com a coluna indexada existe também um nome de dependente e seu sexo, importantes para a localização da informação procurada. Assim são os índices com colunas incluídas, a tabela é indexada por uma coluna (ou um conjunto de colunas), mas junto com o índice estão também outras colunas. Este tipo de índice normalmente é usado para consultas frequentes, onde a coluna indexada é usada no Where e as colunas incluídas são usadas no Select.

Por exemplo:

```
CREATE NONCLUSTERED INDEX Ix_Resp_Dados
    ON Empresa.Dependente (NumSegSocial)
        Include (nomeDependente, sexo)
```

Num eventual Select que buscasse nomeDependente e sexo através de NumSegSocial, o acesso aos dados e a projeção dos campos seria bastante rápida, pois os dados dos campos desejados já estariam disponíveis no próprio índice.

Índices Exclusivos

Determina que os valores de uma determinada coluna devem ser únicos.

Por exemplo, na nossa tabela de Departamentos o índice cluster é o número do departamento. Mas precisamos garantir que o nome do departamento não se repita, e para isso podemos criar um índice não-cluster único com a coluna NomeDepto, usando um comando como abaixo:

```
CREATE UNIQUE INDEX ix_NomeDepto
    ON Empresa.Departamento (NomeDepto);
```

No entanto, lembre-se que já tínhamos definido nomeDepto como um campo Unique no momento de criação da tabela de Departamentos, de forma que o Sql Server automaticamente já criou um índice exclusivo para esse campo. Assim, sempre que definimos um campo como Unique, SQL Server já cria esse tipo de índice, automaticamente.

Veja como ficaram os índices da nossa tabela de Dependentes:

Mantendo a integridade dos índices

Tabelas que sofrem muitas alterações (Insert, Update e Delete) refletem essas modificações nos índices, pois acabam deixando espaços em brancos nas páginas dos mesmos. Estes espaços não utilizados refletem em maior espaço em disco o que acarreta um desperdício de tempo ao percorrer a estrutura do índice.

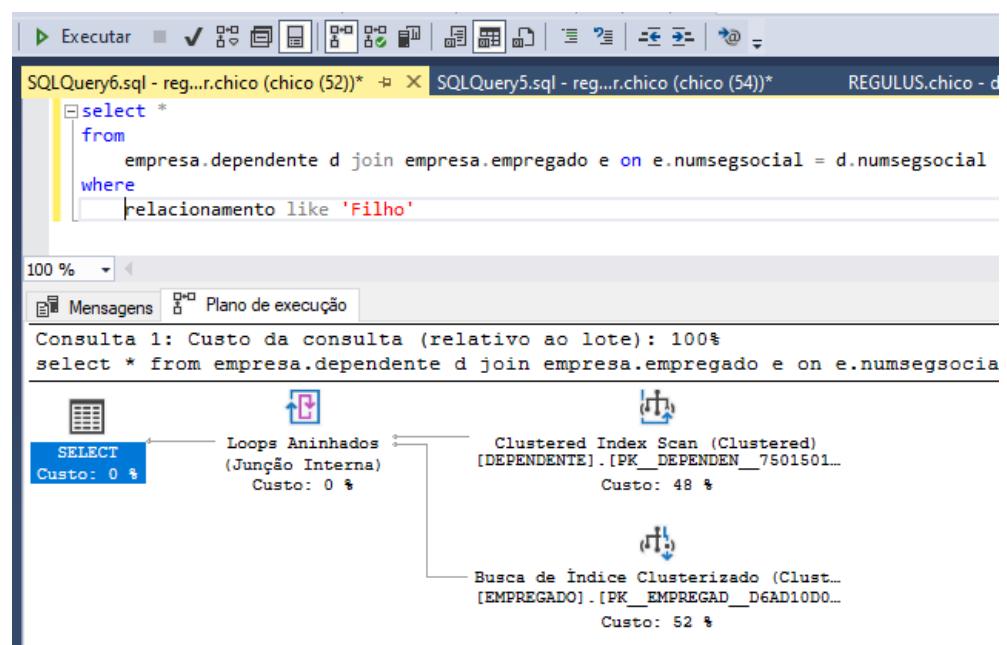
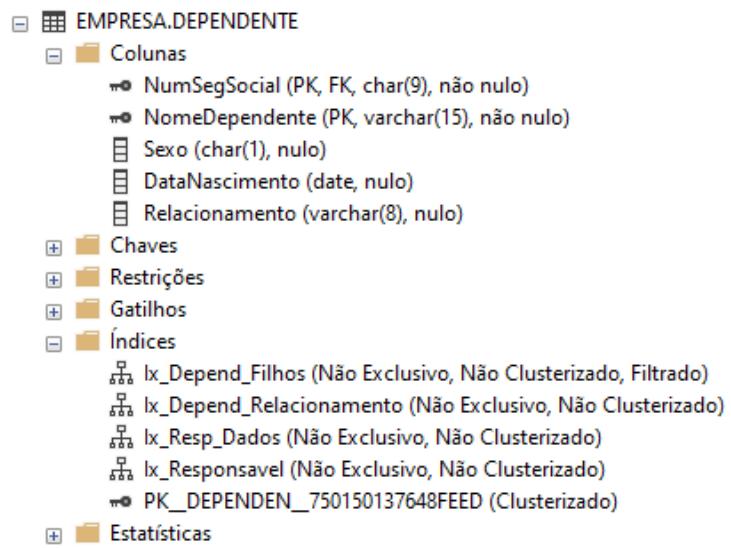
Para resolver esses problemas é necessário manter a integridade dos índices, utilizando os seguintes comandos:

```
ALTER INDEX {nome_indexe | ALL} ON REBUILD
ALTER INDEX {nome_indexe | ALL} ON REORGANIZE
```

A opção REORGANIZE remove somente a fragmentação no nível folha e a opção REBUILD reconstrói todos os níveis do índice.

Plano de execução

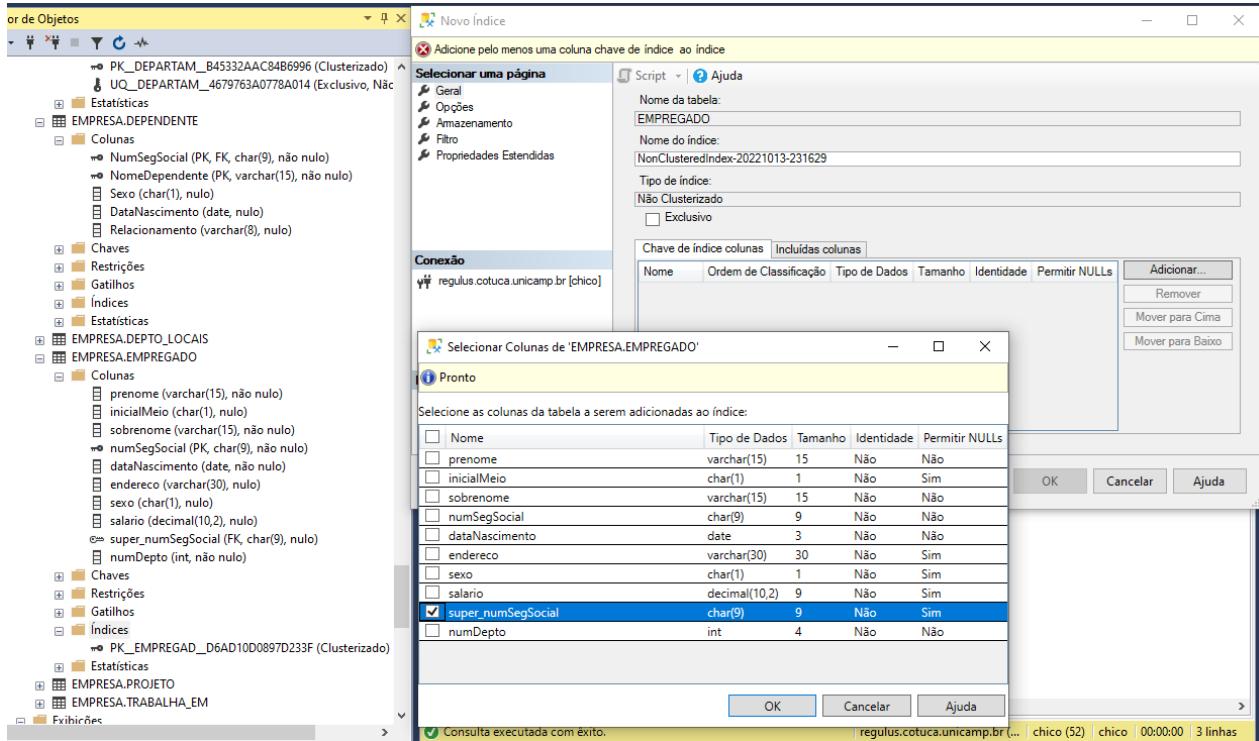
Observe a figura abaixo. Ela mostra o plano de execução da consulta efetuada, buscando os dados de responsáveis e dependentes que sejam filhos. Para exibir o plano de execução, clicamos no botão que está dentro do círculo na figura:



Ferramenta visual para criação de índices

No pesquisador de objetos, você pode criar índices clicando com o botão direito na entrada Índices de uma tabela e selecionar a opção Novo Índice...

A figura a seguir ilustra essa ferramenta, mostrando o resultado de se adicionar um campo para compor a expressão dos índices:



Fontes de consulta:

<https://learn.microsoft.com/pt-br/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver16>
(13/10/2022, 21h53)

<https://learn.microsoft.com/pt-br/sql/relational-databases/performance/display-an-actual-execution-plan?view=sql-server-ver16> (13/10/2022, 23h08)

<https://www.devmedia.com.br/indices-no-sql-server/18353> (13/10/2022, 21h53)

<http://db4beginners.com/blog/indices-no-sql-server/> (13/10/2022, 21h53)

<https://www.jigsawacademy.com/blogs/data-science/types-of-indexes-in-sql-server#:~:text=Indexes%20in%20SQL%20are%20the,through%20each%20row%20of%20it.> (13/10/2022, 21h53)

<https://learn.microsoft.com/pt-br/sql/t-sql/statements/create-view-transact-sql?view=sql-server-ver16>. (18/10/2022, 10h58)

7. Programação no Servidor de Banco de Dados

7.1. Lotes de comandos

O SQL Server e vários outros SGBD possuem uma linguagem de programação interna, que geralmente seguem, em parte, a linguagem SQL padrão aberta. Essa linguagem permite criar lotes de comandos, que são sequências de comandos que acessam o banco de dados, permitindo desenvolver algoritmos que processam os dados acessados. No Sql Server, essa linguagem é chamada de Transact-SQL ou T-SQL. Em Oracle, temos a PL/SQL.

Os principais comandos são destinados à criação de procedimentos e funções (semelhantes aos métodos de C# e Java), declaração de variáveis, atribuição de valores a elas, controle de fluxo e processamento de cursores, que são resultados de consultas e podem ser processados como se fossem arquivos de dados.

Em geral, um lote de comandos em T-SQL inicia com uma instrução de declaração do lote, um BEGIN e termina com a instrução END; entre elas codificaremos os demais comandos que esse lote deverá executar. Portanto, um lote de comandos contém um ou mais comandos T-SQL delimitados por BEGIN e End. Um lote de comandos é enviado ao servidor de banco de dados como uma única unidade. O Servidor procurará analisar os comandos e executá-los. As variáveis e parâmetros declarados em um lote de comandos são sempre locais, não sendo acessíveis externamente ao lote. Em geral, a palavra GO finaliza um lote e informa ao Sql Server o final do lote.

Quando um lote é enviado por um cliente, como quando você pressiona o botão Executar no SSMS, ele é analisado quanto a erros de sintaxe pelo mecanismo do SQL Server. Os erros encontrados farão com que todo o lote seja rejeitado; não haverá nenhuma execução parcial de instruções dentro do lote.

Se o lote passar na verificação de sintaxe, o SQL Server executará outras etapas, resolvendo nomes de objeto, verificando permissões e otimizando o código para execução. Quando esse processo for concluído e a execução for iniciada, as instruções serão bem-sucedidas ou falharão individualmente. Esse é um contraste importante com relação à verificação de sintaxe. Quando ocorre um erro de execução em uma linha de comando, a próxima linha de comando pode ser executada, a menos que você tenha adicionado tratamento de erro ao código.

7.2. Declaração de Variáveis

Da mesma forma que em qualquer linguagem de programação, em T-SQL as variáveis são usadas para armazenar valores que serão posteriormente usados. Em T-SQL elas devem ser declaradas antes de seu uso. No momento da declaração já podemos atribuir um valor a elas, se o tivermos disponível. Usamos a instrução DECLARE para declarar variáveis, como vemos abaixo. Cada variável, no T-SQL, tem seu nome iniciado com @ e deve ter também um tipo de dados.

```
DECLARE      @quantosRegistros int = 1,
            @categoria   int,
            @nome        nvarchar(200) = 'Maria';
```

T-SQL não diferencia letras maiúsculas de minúsculas. As variáveis precisam ser declaradas no mesmo lote em que são referenciadas. Em outras palavras, todas as variáveis T-SQL são locais no escopo do lote, tanto na visibilidade quanto no tempo de vida. Somente outras instruções no mesmo lote podem ver uma variável declarada no lote. Uma variável é destruída automaticamente quando o lote termina.

Podemos atribuir um valor a uma variável logo em sua declaração, como foi feito no exemplo acima para as variáveis @quantosRegistros e @nome. Caso desejemos atribuir valores depois da declaração, podemos usar o comando SET ou o comando SELECT, como vemos nos exemplos a seguir:

```

BEGIN
    DECLARE      @quantosRegistros int = 1, @categoria int,
                @nome          nvarchar(200) = 'Maria';

    set @categoria = 2
    set @nome = @nome + N' tinha um carneirinho' -- N indica Unicode
    declare @idMedico int,
            @dataConsulta date,
            @horaInicio time,
            @horaFim time
    select
        @idMedico = idMedico, @dataConsulta = dataConsulta,
        @horaInicio = horaInicio, @horaFim = horaFim
    from
        cons.Consulta
    Where
        idConsulta = 1

    print 'Medico '+cast(@idMedico as varchar)
    print 'data '+Cast(@dataConsulta as varchar)
    print 'inicio '+Cast(@horaInicio as Varchar)
    print 'fim   '+Cast(@horaFim as Varchar)
    print @nome

END

```

Acima, usamos Select para efetuar uma busca na tabela Consulta do esquema Cons e armazenamos, nas variáveis indicadas, os valores dos campos retornados. Em seguida, os valores dessas variáveis foram convertidos para strings (varchar) e concatenados com uma string de título, e escritos na tela com o comando print.

7.3. Comandos de Controle de Fluxo de Execução

O SQL Server fornece elementos de linguagem que controlam o fluxo de execução do programa em lotes T-SQL, procedimentos armazenados e funções definidas pelo usuário de várias instruções. Esses elementos de controle de fluxo significam que você pode determinar programaticamente se deseja executar instruções e determinar programaticamente a ordem dessas instruções que devem ser executadas.

Esses elementos incluem, entre outros:

- IF...ELSE, que executa um código baseado em uma expressão booliana.
- WHILE, que cria um loop que é executado desde que uma condição seja true.
- BEGIN...END, que define uma série de instruções T-SQL que devem ser executadas juntas.
- Outras palavras-chave, por exemplo, são BREAK, CONTINUE, WAITFOR e RETURN, usadas para dar suporte a operações de controle de fluxo T-SQL.

Confira um exemplo de uma instrução IF:

```

IF OBJECT_ID('dbo.t1') IS NOT NULL
    DROP TABLE dbo.t1
GO

```

Object_Id() é uma função embutida que retorna a identificação de um objeto qualquer do banco de dados. Se não existir, retorna NULL. No If acima, se o resultado de Object_Id() não é nulo, significa que existe a tabela cujo nome foi passado como parâmetro e, assim, ele é removido (DROP Table).

Da mesma forma que em C# e outras linguagens, quando colocamos um único comando dentro de um IF, ELSE ou WHILE, não é obrigatório o uso dos delimitadores BEGIN-END. Quando houver mais de um comando subordinado, deverão ser delimitados por BEGIN-END.

```
IF OBJECT_ID('Cons.Paciente') IS NULL -- este objeto não existe no BD
BEGIN
    PRINT 'O objeto especificado não existe.';
END
ELSE
BEGIN
    PRINT 'O objeto especificado existe.';
END;
```

Nas operações de manipulação de dados, usar IF com a palavra-chave EXISTS pode ser uma abordagem útil para verificações de existência eficientes, como no seguinte exemplo:

```
IF EXISTS (SELECT * FROM Cons.Consulta WHERE idMedico = 5)
BEGIN
    PRINT 'Médico 5 possui consultas marcadas.';
END;
```

A instrução WHILE é usada para executar o código em um loop com base em uma condição. Como a instrução IF, a instrução WHILE determina se a instrução ou o bloco seguinte (se BEGIN..END for usado) é executado. O loop continua a ser executado, desde que a condição seja avaliada como TRUE. Normalmente, o loop é controlado com uma variável testada pela condição e manipulada no corpo do próprio loop.

O seguinte exemplo usa a variável @idMedico na condição e altera o valor dela no bloco BEGIN...END:

```
DECLARE @idMedico AS INT = 1, @sNome AS NVARCHAR(20);
WHILE @idMedico <= 100
BEGIN
    SELECT @sNome = sobrenomeMed
    FROM Cons.Medico
    WHERE idMedico = @idMedico;
    PRINT @sNome;
    SET @idMedico += 1;
END;
```

Para obter opções adicionais em um loop WHILE, você pode usar as palavras-chave CONTINUE e BREAK para controlar o fluxo, de forma semelhante a C# e Java.

Exercício prático

Use sua conta da Microsoft na Unicamp (seuRA@m.unicamp.br) e senha para acessar o laboratório de programação, crie seu perfil e realize as instruções dadas.

<https://learn.microsoft.com/pt-br/training/modules/get-started-transact-sql-programming/6-exercise-program-transact-sql>

Fontes de consulta:

<https://learn.microsoft.com/pt-br/training/modules/get-started-transact-sql-programming/2-describe-transact-sql-for-programming> (14/10/2022 19h53)

7.4. Triggers

Um Trigger (gatilho) é um objeto associado a uma tabela que define ações a serem executadas automaticamente quando certos eventos e condições ocorrerem.

Essa funcionalidade está ligada ao conceito de **Banco de Dados Ativo**, que é um sistema de banco de dados que, além das funcionalidades normais inerentes a um SGBD, permite definir um conjunto de regras que estabelece comportamentos automáticos. Esse sistema é capaz de reconhecer eventos, ativar as regras correspondentes e, quando uma condição especificada é verdadeira, executa as ações que a ela correspondam.

Tais regras são conhecidas como regras ECA devido ao paradigma Evento-Condição-Ação – ilustrado na figura abaixo – que baliza seu princípio de funcionamento que, inclusive, é bastante intuitivo. Quando um *evento* que altera o estado do banco de dados ocorre – linhas acrescentadas ou modificação de uma coluna, por exemplo –, a *condição* de execução da regra é avaliada e, sendo verdadeira, a *ação* é executada.



Visão esquemática das regras ECA.

Tomando como exemplo a necessidade de manutenção do estoque de uma indústria, poderíamos criar regras ECA que monitorassem os níveis de disponibilidade das matérias primas. Na identificação de um nível baixo – o evento – de uma das matérias-primas críticas para a indústria – a condição –, a compra seria disparada automaticamente ao fornecedor sem qualquer intervenção humana – a ação.

Para implementar esse paradigma, SQL Server e alguns outros SGBDs como Oracle, Informix, MySQL, possuem o mecanismo de Trigger. Um **Trigger** é bloco de comandos **SQL** que é automaticamente executado quando um comando **INSERT**, **DELETE** ou **UPDATE** for executado em uma tabela do banco de dados.

Os triggers podem ser disparados para responderem antes ou depois de um evento. São usados para realizar tarefas relacionadas com validações, restrições de acesso, rotinas de segurança e consistência de dados; desta forma esses controles deixam de ser executados pela aplicação e passam a ser executados no Servidor de Banco de Dados através dos Triggers em determinadas situações:

- Mecanismos de validação envolvendo múltiplas tabelas
- Criação de conteúdo de uma coluna derivada de outras colunas da tabela
- Realizar análise e atualizações em outras tabelas com base em alterações e/ou inclusões da tabela atual

A criação de um Trigger envolve duas etapas:

1. Um comando SQL que vai disparar o Trigger (**INSERT**, **DELETE**, **UPDATE**)
2. A ação que o Trigger vai executar (geralmente um bloco de códigos SQL)

Para criar um Trigger usamos o comando abaixo:

CREATE TRIGGER [NOME DO TRIGGER]

```

ON [NOME DA TABELA]
[FOR/AFTER/INSTEAD OF] [INSERT/UPDATE/DELETE]
AS
--COMANDOS DO TRIGGER

```

Como exemplo, vamos pensar no nosso Banco de Dado do Consultório Médico. Para incluirmos ou alterarmos uma consulta corretamente, sem que haja sobreposição de horários no mesmo dia, podemos usar um Trigger executado antes do Insert e do Update. Nesse trigger, devemos fazer uma consulta que nos diga se existe algum registro de consulta do mesmo médico, no mesmo dia e em horários que choquem com o da consulta que desejamos registrar. Caso haja, enviamos uma mensagem de erro ao programa que solicitou e o programa terá de detectar esse erro e cancelar a inclusão ou alteração.

Antes disso, vamos alterar os campos horaInício e horaFim de tipo DateTime para Time, usando os comandos abaixo:

```

alter table cons.consulta
    alter column dataConsulta Date not null
alter table cons.consulta
    alter column horaInicio time not null
alter table cons.consulta
    alter column horaFim time not null

```

The screenshot shows a Microsoft SQL Server Management Studio window. At the top, it displays the database name 'REGULUS.chico - cons.Consulta' and the query name 'SQLQuery5.sql - reg...r.chico (chico (54))'. Below this is a table with the following data:

	idConsulta	idMedico	idPaciente	dataConsulta	horaInício	horaFim	observacoes	ativo
	1	4	1	2022-10-05 11:31:49.000	16:00:00	16:15:00	Paciente levará exames que...	True
	3	2	2	2022-10-28 11:31:49.000	17:00:00	17:15:00	NULL	NULL
	4	4	5	2022-10-20 11:31:49.000	12:30:00	13:00:00	Paciente necessita atestado	True
	5	1	3	2022-10-24 11:31:49.000	NULL	NULL	NULL	NULL
...	NULL	4	2	2022-10-15 00:00:00.000	16:05:00	18:00:00	teste trigger	True
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

A message dialog box from 'Microsoft SQL Server Management Studio' is displayed, stating: 'Nenhuma linha foi atualizada.' (No rows were updated.) It also includes error details: 'Os dados da linha 5 não foram confirmados.', 'Origem do Erro: .Net SqlClient Data Provider.', 'Mensagem de Erro: Conflito de horário', and 'A transação foi encerrada no gatilho. O lote foi anulado.' (The transaction was rolled back at the trigger. The batch was canceled.) At the bottom of the dialog are 'OK' and 'Ajuda' buttons.

```

CREATE TRIGGER [cons].[trVerificarHorario]
ON [cons].[Consulta]
for INSERT, UPDATE AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @ultimoid int = @@Identity -- obtém idConsulta recém inserido
    print 'Último id: '+cast(@ultimoid as varchar)
    declare @idConsulta int, @idMedico int, @idPaciente int, @idPaciente2 int,

```

```

@idMedico2 int, @idConsulta2 int,
@dataConsulta date, @dataconsulta2 date,
@horalInicio time, @horalInicio2 time,
@horaFim time, @horaFim2 time
select @idConsulta = idConsulta, @idMedico = idMedico,
       @idPaciente = idPaciente, @dataConsulta = dataConsulta,
       @horalInicio = horalInicio, @horaFim = horaFim
from Inserted
print 'Consulta recém-inserida:'
print 'Id Cons '+cast(@idConsulta as varchar)
print 'Medico '+cast(@idMedico as varchar)
print 'Pac. '+cast(@idPaciente as varchar)
print 'data '+Cast(@dataConsulta as varchar)
print 'inicio '+Cast(@horalInicio as Varchar)
print 'fim '+Cast(@horaFim as Varchar)

SELECT top 1 @idConsulta2 = isnull(idConsulta,0), @idMedico2 = idMedico,
       @idPaciente2 = idPaciente, @dataconsulta2 = dataConsulta,
       @horalInicio2 = horalInicio, @horaFim2 = horaFim
FROM cons.Consulta
WHERE (idMedico = @idMedico) AND (idConsulta <> @ultimold) and
      --(idPaciente <> @idPaciente) and
      (dataConsulta = @dataConsulta) AND
      ( (@horalInicio >= horalInicio and @horalInicio <= horaFim) OR
        (@horaFim >= horalInicio AND @horaFim <= horaFim)
      )
order by idMedico, dataConsulta, horalInicio, horaFim

if @idConsulta2 <> 0
begin
    print 'Consulta anterior em conflito de horário com a recém-inserida:';
    print 'Consulta '+cast(@idConsulta2 as varchar)
    print 'Medico2 '+cast(@idMedico2 as varchar)
    print 'Pac2. '+cast(@idPaciente2 as varchar)
    print 'data2 '+Cast(@dataConsulta2 as varchar)
    print 'inicio2 '+Cast(@horalInicio2 as Varchar)
    print 'fim 2 '+Cast(@horaFim2 as Varchar)
    Delete from cons.consulta where idConsulta = @idConsulta
    RAISERROR('Conflito de horário', 15, 1);
END
else
    print 'horario valido'
END;

```

Fontes de consulta:

- <https://lis-unicamp.github.io/wp-content/uploads/2014/09/adb-24j.pdf> (14/10/2022, 14h45)
- <https://www.devmedia.com.br/bancos-de-dados-ativos-revista-sql-magazine-94/23025#> (14/10/2022, 14h52)
- <https://www.devmedia.com.br/triggers-no-sql-server-teoria-e-pratica-aplicada-em-uma-situacao-real/28194>
- <https://www.sqlservertutorial.net/sql-server-triggers/sql-server-create-trigger/>
- https://www.macoratti.net/sql_trig.htm
- <https://learn.microsoft.com/pt-br/training/modules/get-started-transact-sql-programming/>

7.5. Procedimentos Armazenados

Um procedimento armazenado é um conjunto de comandos SQL que manipulam tabelas e dados, de maneira programada.

Essa programação busca implantar as regras de negócio no banco de dados, de maneira que o processamento de grandes quantidades de dados seja feito no backend, sem aumentar o tráfego da rede, como ocorreria se os dados tivessem de ser trazidos para a máquina do cliente processá-los localmente.

Além disso, com as regras de negócio centralizadas no servidor de BD, qualquer alteração efetuada nas mesmas é disponível para os clientes automaticamente.

Um Stored Procedure é um lote de um ou mais comandos SQL que são armazenados no servidor de banco de dados. Stored Procedures são semelhantes a funções por serem blocos de lógica encapsulados que podem aceitar dados (através de parâmetros de entrada) e retornar dados (através de parâmetros de saída e de result sets, que são os registros resultantes de um select). Stored procedures tem várias vantagens:

- São fáceis de manter atualizadas: por exemplo, você pode otimizar os comandos de uma stored procedure sem recompilar as aplicações que a utilizam. Elas também padronizam a lógica de acesso aos dados em um único lugar -- o banco de dados -- tornando mais fácil para diferentes aplicações reutilizarem essa lógica de uma maneira consistente (em termos de orientação da objetos, stored procedures definem a interface do seu banco de dados)
- Permitem implementar uso mais seguro do banco de dados: por exemplo, você pode permitir à conta que roda a sua aplicação usar certos stored procedures mas restringir o acesso direto às tabelas subjacentes usando essa conta. Assim, o usuário do site apenas terá como executar os stored procedures cujo código já é testado e confiável, sem ter acesso direto a comandos Insert, Update, Delete, Drop, etc nas tabelas.
- Elas podem aumentar o desempenho: Pelo fato de uma stored procedure agrupar múltiplos comandos, uma única "viagem" ao banco de dados através da rede permite que muito trabalho seja executado nesse servidor. Se seu banco de dados está em outro computador, isso reduz dramaticamente o tempo total necessário para realizar tarefas complexas, que passam a ser feitas em lote nas stored procedures.
- SQL Server precompila todos os comandos SQL, incluindo aqueles que você monta dinamicamente em suas aplicações. Isso significa que você obtém os benefícios da compilação mesmo que não use stored procedures. No entanto, stored procedures ainda tendem a aumentar os benefícios de desempenho, porque sistemas que as usam tendem a ter menos variabilidade. Sistemas que usam instruções SQL dinâmicas tendem a usar comandos diferentes para fazer tarefas semelhantes, o que significa que planos de execução já compilados não podem ser reutilizados efetivamente.

Para criar um stored procedure (SP), usamos o comando SQL create procedure, cuja sintaxe é:

```
create or alter procedure <esquema>.<nome do procedimento>
    [<declaração de parâmetros>]
    as
        <comandos>
```

Exemplo:

```
create or alter procedure Empresa.spContaLocais
    @numDepto int,                      -- parâmetro por valor, de entrada
    @quantos int output                 -- parâmetro por referência, de saída
    as begin
        Select @quantos = count(*) from Empresa.Depto_Locais
        where numDepto = @numDepto
    end
```

Esse procedimento ficará armazenado no servidor de Banco de Dados e, quando for necessário executar uma contagem de locais associados a algum departamento do esquema EMPRESA, bastará que a aplicação chame esse procedimento e informe, via parâmetro, o número do departamento cuja contagem de locais é desejada.

O resultado dessa contagem é devolvido no parâmetro @quantos. Isso é indicado pelo uso da palavra reservada output que, semelhante ao ref ou out de C#, indica que o parâmetro é modificado e poderá devolver um valor para a aplicação.

Cada servidor de BD (Oracle, SQL Server, Sybase, DB2, etc) possui uma sintaxe específica dos comandos SQL para Stored Procedures, de modo que poderá ser difícil portar o código de procedimentos escritos em um servidor para outro servidor diferente.

Para executar uma Stored Procedure, a aplicação precisa chamar esse procedimento, usando o comando EXEC e passando os parâmetros. Para o procedimento acima, podemos chama-lo usando os comandos abaixo. Note que declaramos uma variável para ser passada como parâmetro output, além de exibir o valor retornado:

```
declare @contagem int = 0
exec empresa.spContaLocais 1, @contagem output
print 'O departamento 1 tem '+cast(@contagem as varchar)+' local(is)'
```

Para o nosso banco de dados do esquema EMPRESA, esse comando resulta no seguinte:

```
Mensagens
O departamento 1 tem 1 locais

Horário de conclusão: 2022-10-25T11:44:54.4064930-03:00
```

Podemos executar o mesmo procedimento mudando o valor do parâmetro @numDept, como vemos abaixo, quando passamos o departamento 5 como parâmetro de entrada e recebemos o valor 3 no parâmetro de saída:

```
declare @contagem int = 0
exec empresa.spContaLocais 5, @contagem output
print 'O departamento 5 tem '+cast(@contagem as varchar)+' local(is)'
```

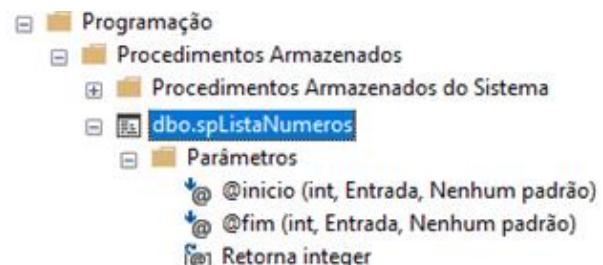
```
Mensagens
O departamento 5 tem 3 local(is)

Horário de conclusão: 2022-10-25T11:46:32.6343436-03:00
```

Um procedimento armazenado pode chamar outro, usando o comando exec em seu código interno, passando os parâmetros necessários e os tratamentos no retorno dessa chamada. Vamos criar um procedimento armazenado em nosso banco de dados. Numa janela de consulta do SSMS digite o seguinte:

```
create or alter procedure spListaNumeros @inicio int, @fim int
as
begin
    declare @iContador int
    set @iContador = @inicio -- ou: select @iContador = @inicio
    while @iContador <= @fim
    begin
        print @iContador
        set @iContador = @iContador + 1
    end
end
```

Execute essa consulta para criar o procedimento armazenado. Observe no Pesquisador de Objetos o item Programação. Dentro dele temos um subitem Procedimentos Armazenados onde encontraremos esse procedimento recém-criado:



Usamos o comando EXEC para chamar esse procedimento, passando os valores dos parâmetros @inicio e @fim, como vemos abaixo. Como esses parâmetros são somente de entrada, podemos passar valores constantes. Se fossem parâmetros de entrada, deveríamos usar uma variável para receber os resultados.

Tratamento de Exceções

Uma prática recomendada de abordagem nos comandos de manutenção dos dados das tabelas de um banco de dados é encapsulá-los em Stored Procedures. Nessa abordagem, os comandos Insert, Update e Delete não são executados diretamente pela aplicação mas, ao contrário, a aplicação chama um Stored Procedure especializado em inserção (por exemplo) para inserir dados de uma determinada tabela. Assim, para cada tabela, teremos uma stored procedures para a ação de inserir, outro stored procedure para a ação de alterar e um terceiro para a ação de remover registros dessa tabela. Pode parecer redundância de código, mas essa abordagem permite que sempre que uma dessas ações for executada, sejam verificados pelo servidor condições de integridade dos dados, evitar e/ou acusar erros antes que eles sejam armazenados nas tabelas, ou que se impeçam remoções que poderiam quebrar a consistência do banco de dados.

Como exemplo, estudemos os códigos das Stored Procedures abaixo:

The screenshot shows the SSMS interface with the following details:

- SQLQuery5.sql - reg...r.chico (chico (54))*


```
exec spListaNumeros 22, 45
```
- Mensagens pane showing the output of the query:


```
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
```
- Horário de conclusão: 2022-10-2

```
CREATE or ALTER PROCEDURE Empresa.spInserirEmpregado
    @prenome varchar(15), @inicialMeio char(1), @sobrenome varchar(15),
    @numSegSocial char(9), @dataNascimento date, @endereco varchar(30),
    @sexo char(1), @salario decimal, @super_numSegsocial int,
    @numDept int

AS
BEGIN
    if exists(select numdepto from empresa.DEPARTAMENTO
              where numDept = @numDept)
        begin
            insert into empresa.EMPREGADO values
                (@prenome, @inicialMeio, @sobrenome, @numSegSocial, @dataNascimento,
                 @endereco, @sexo, @salario, @super_numSegSocial, @numDept)
            if @@error <> 0 -- inserção não realizada
                begin
                    declare @Mensagem nvarchar
                    Select @Mensagem = Error_Message()
                    Raiserror('Erro na inclusão desse empregado: %s', @Mensagem, 16, 1)
                end
        end
    else
        Raiserror('Número de departamento inexistente. Corrija.', 16, 2)
```

```
END
```

Acima, **if exists()** verifica se existe resultado no select que busca o registro de departamento cuja chave primária (numDepto) é igual ao valor @numDepto, passado como parâmetro pela aplicação.

Se existir, efetua o comando Insert na tabela Empregado, usando os valores dos parâmetros na mesma sequência dos campos dessa tabela, para que esse novo registro seja inserido na tabela.

Se, nessa inclusão, ocorrer algum erro, a variável de sistema @@error ficará diferente de 0, indicando que ocorreu um erro. Nesse caso, o segundo if captura esse erro e dispara uma exceção ao chamar o procedimento Raiserror. Antes de chamar Raiserror, buscou-se a mensagem de erro, a severidade desse erro e o estado (sobre os quais logo estudaremos).

Executando essa stored procedure para incluir um registro no departamento de número 99, teremos um erro, já que esse departamento não existe na tabela Departamento:

```
exec empresa.spInserirEmpregado 'João', 'C', 'Silva', '123654987', '2000-10-26',
'Rua das Camélias, 17', 'M', 10000, '123456789', 99
```

Por outro lado, executando o procedimento armazenado para um novo empregado no departamento 1, não haverá erros e o registro será incluído:

```
exec empresa.spInserirEmpregado 'João', 'C', 'Silva', '123654987', '2000-10-26',
'Rua das Camélias, 17', 'M', 10000, '123456789', 1
```

Poderíamos criar um procedimento armazenado alternativo para inclusão, no qual passamos o nome do departamento e não seu número, e o procedimento se encarrega de descobrir o número do departamento correspondente a esse nome para, logo em seguida, inserir o registro do novo empregado. O procedimento armazenado spInserirEmpregadoComDepartamento se encarrega disso:

```
CREATE or ALTER PROCEDURE Empresa.spInserirEmpregadoComDepartamento
@prenome varchar(15), @inicialMeio char(1), @sobrenome varchar(15),
@numSegSocial char(9), @dataNascimento date, @endereco varchar(30),
@sexo char(1), @salario decimal, @super_numSegsocial int,
@nomeDepto varchar(15)
AS
BEGIN
declare @numDepto int = 0
select @numDepto = numdepto from empresa.DEPARTAMENTO
where nomeDepto = @nomeDepto
if @numDepto <> 0
begin
insert into empresa.EMPREGADO values
(@prenome, @inicialMeio, @sobrenome, @numSegSocial,
```

```

        @dataNascimento, @endereco, @sexo, @salario, @super_numSegSocial,
        @numDepto)
    if @@error <> 0 -- inserção não realizada
    begin
        declare @Mensagem nvarchar
        Select @Mensagem = Error_Message()
        Raiserror('Erro na inclusão desse empregado: %s', @Mensagem, 16, 1)
    end
end
else
    Raiserror('Nome de departamento inexistente. Corrija.', 16, 2)
END

```

Executando, temos os exemplos abaixo:

```

exec empresa.spInserirEmpregadoComDepartamento 'Márcia','T','Correia','183654287',
                                                '2000-10-26', 'Rua das Camélias, 17', 'F',
                                                10000, '123456789', 'PESQUISA'

```

(1 linha afetada)

```

Horário de conclusão: 2022-10-25T14:22:16.8614566-03:00
exec empresa.spInserirEmpregadoComDepartamento 'Sérgio','M','Marques','813654278',
                                                '1969-10-26', 'Rua dos Gerânicos, 142 ap 5', 'M',
                                                10000, '123456789', 'VENDAS'

```

Msg 50000, Nível 16, Estado 2,
Procedimento empresa.spInserirEmpregadoComDepartamento,
Linha 28 [Linha de Início do Lote 49]
Nome de departamento inexistente. Corrija.

Horário de conclusão: 2022-10-25T14:24:50.8622577-03:00

Se incluirmos um empregado com numSegSocial repetido, o procedimento acusará o erro e não o incluirá.

Observe que as mensagens de erro exibidas são relativamente completas, informando o nome do procedimento armazenado onde ocorreram e a linha do código. Sobre os parâmetros de Raiserror, podemos dizer o seguinte:

O **primeiro parâmetro, Message**, é uma string com a mensagem de erro que se deseja enviar para a aplicação exibir.

O **segundo parâmetro, Error Severity** é um inteiro que fornece informação sobre a severidade do erro que ocorreu. Assim, SQL Server trata esse valor nas faixas de importância abaixo:

- De 0 a 10 – mensagens com informações simples
- De 11 a 16 – erros que são considerados como podendo ser solucionados pelo próprio usuário
- 17 a 19 – são erros não-fatais nos recursos do SQL Server, seus recursos, mecanismo de funcionamento e outros que não levem ao cancelamento da aplicação.
- 20 a 25 – são Erros Fatais, que causam o término da execução do próprio mecanismo de funcionamento do Sql Server imediatamente, ou seja, o programa do servidor de banco de dados deixa de ser acessível e utilizável.

O terceiro parâmetro, Error State, é um inteiro que enumera os erros detectados, de forma que permite localizar exatamente o local do seu código onde o erro foi detectado no seu código. Imagine que você tenha uma Stored Procedure extensa, com centenas ou milhares de linhas e que você está detectando e tratando erros em vários lugares diferentes desse código. O número de Error State poderá lhe ajudar a saber onde o erro foi efetivamente detectado, se você fornecer um número de Error State diferente para cada momento em que detecta erros.

Embora Raiserror seja bastante fácil de usar, as novas aplicações devem usar outra estrutura para detecção e tratamento de erros. É a estrutura Try-Catch, semelhante ao que já estudamos em outras linguagens de programação. Os comandos de um Try devem ficar entre os comandos delimitadores BEGIN TRY e END TRY. Já o Catch inicia logo após isso, com os comandos delimitadores BEGIN CATCH e END CATCH.

O procedimento armazenado anterior foi modificado para usar Try-Catch e Throw:

```
CREATE or ALTER PROCEDURE Empresa.spInserirEmpregadoComDept_Throw
    -- Add the parameters for the stored procedure here
    @prenome varchar(15), @inicialMeio char(1), @sobrenome varchar(15),
    @numSegSocial char(9), @dataNascimento date, @endereco varchar(30),
    @sexo char(1), @salario decimal, @super_numSegsocial int,
    @nomeDepto varchar(15)
AS
BEGIN
    declare @numDepto int = 0
    select @numdepto = numdepto
    from empresa.DEPARTAMENTO
    where nomeDepto = @nomeDepto
    if @numDepto <> 0
        begin try
            insert into empresa.EMPREGADO values
                (@prenome, @inicialMeio, @sobrenome, @numSegSocial,
                 @dataNascimento, @endereco, @sexo, @salario, @super_numSegSocial,
                 @numDepto)
        end try
        begin catch
            declare @Mensagem nvarchar(2048)
            set @Mensagem = 'Erro: '+Error_Message();
            throw 51200, @Mensagem, 1
        end catch
        else
            Throw 51220, 'Nome de departamento inexistente. Corrija.', 2
    END
```

Após criar esse novo procedimento armazenado, ele foi testado com o comando abaixo:

```
exec empresa.spInserirEmpregadoComDept_Throw 'Sérgio', 'M', 'Marques', '183654287',
                                                '1969-10-26', 'Rua dos Gerânios, 142 ap 5', 'M',
                                                10000, '123456789', 'SEDE'
```

```
(0 linhas afetadas)
Msg 51200, Nível 16, Estado 1, Procedimento empresa.spInserirEmpregadoComDept_Throw,
Linha 20 [Linha de Início do Lote 47]
Erro: Violação da restrição PRIMARY KEY 'PK_EMPREGAD_D6AD10D0897D233F'.
Não é possível inserir a chave duplicada no objeto 'EMPRESA.EMPREGADO'.
O valor de chave duplicada é (183654287).

Horário de conclusão: 2022-10-25T19:46:07.3302734-03:00
```

Por outro lado, se fizermos o próprio teste num bloco Try-Catch, obteremos uma resposta mais “limpa”:

```
begin try
exec empresa.spInserirEmpregadoComDept Throw 'Sérgio', 'M', 'Marques', '183654287',
                                              '1969-10-26', 'Rua dos Gerânicos, 142 ap 5', 'M',
                                              10000, '123456789', 'SEDE'
end try
begin catch
    declare @Mensagem nvarchar(2048)
    Select @Mensagem = Error_Message();
    print @mensagem
end catch
```

```
(0 linhas afetadas)
Erro: Violação da restrição PRIMARY KEY 'PK_EMPREGAD_D6AD10D0897D233F'.
Não é possível inserir a chave duplicada no objeto 'EMPRESA.EMPREGADO'.
O valor de chave duplicada é (183654287).

Horário de conclusão: 2022-10-25T19:49:23.4260237-03:00
```

Levando em consideração essas estratégias que discutimos sobre procedimento armazenados, é importante que, quando você estiver desenvolvendo uma aplicação que acesse bancos de dados, se preocupe em desenvolver procedimentos armazenados para as principais operações, como inclusão, exclusão, atualização e, em certas situações, para consultas importantes.

Esses procedimentos ficam armazenados no servidor de banco de dados e, assim, qualquer modificação feita nos mesmos será prontamente disponibilizada para todos os usuários clientes da sua aplicação. Também permitirá que menos dados trafeguem pela rede, pois as regras de negócio poderão ser codificadas no próprio servidor, evitando, assim, que dados tenham de vir do servidor a cada cliente e que este tenha de encaminhar os resultados ao servidor.

Dessa forma, o tráfego em rede diminui, o esforço de codificação das regras de negócios fica focado em um único lugar (que chamamos de back-end), usando uma única linguagem de programação (a SQL do servidor).

Por outro lado, se a necessidade de processamento que o uso da aplicação for muito alta, e esse processamento for totalmente transferido para o servidor de banco de dados, este poderá ficar sobrecarregado ou exigir constantes atualizações de memória, processador e unidades de armazenamento, bem como de infraestrutura de rede.

Com a crescente conectividade das aplicações, a abordagem cliente-servidor tem se utilizado de diferentes estratégias para evitar que o servidor de banco de dados fique estrangulado e não possa atender à sua principal função, que é garantir armazenamento e consulta eficientes aos dados.

Estudaremos essas estratégias, como a computação distribuída ou em camadas, logo mais.

Além dessas vantagens, o uso de stored procedures com parâmetros permite ao servidor “sanitizar” os dados, de forma que se evite a ocorrência de ataques de “Injeção de SQL” (SQL Injection), nos quais usuários maliciosos digitam comandos SQL em campos de entrada de dados na aplicação que, devido à forma como são digitados, poderão destruir dados se os comandos Insert, Delete ou Update fossem executados pela aplicação. Na próxima sessão veremos mais sobre isso.

7.6. SQL INJECTION

Até este momento, todos os exemplos que vimos usaram comandos SQL codificados diretamente nos arquivos de código. Isto torna o exemplo mais simples, direto e relativamente seguro.

Isso também significa que esses exemplos não são realistas e eles não demonstram um dos mais sérios riscos para aplicações web (ou mesmo desktop e mobile) que interajam com um servidor de bancos de dados - ataques SQL Injection ou "Injeção de SQL".

Em termos simples, SQL Injection é o processo de passar código SQL para uma aplicação, de uma forma que não foi pretendida ou antecipada pelos seus desenvolvedores. Isto pode se tornar possível devido a um design deficiente da aplicação e ele afeta somente aplicações que usam técnicas de construção de strings SQL para criar um comando com valores fornecidos pelo usuário em TextBoxes, por exemplo.

Considere o exemplo exibido na figura abaixo. Nesse exemplo, o usuário digita um ID de cliente e o GridView apresenta todas as linhas de dados desse cliente. Num exemplo mais realista, o usuário também precisaria fornecer algum tipo de informação de autenticação, como uma senha. Ou, o ID do usuário poderia ser baseado em uma tela anterior de login e o TextBox permitiria ao usuário fornecer critérios adicionais, tais como um intervalo de datas ou o nome do produto em um pedido.

CustomerID	OrderID	Items	Total
ALFKI	10643	3	1086.0000
ALFKI	10692	1	879.0000
ALFKI	10702	2	330.0000
ALFKI	10935	2	951.0000
ALFKI	10952	2	491.2000
ALFKI	11011	2	960.0000

O problema dessa abordagem é como esse comando SQL é executado. Neste exemplo, o comando SQL é construído dinamicamente usando a técnica de construção de string (chamada de SQL Dinâmico). O valor do campo txtId é, simplesmente, copiado no meio da string. Eis o código:

```

string cadConexao = "Data Source=Regulus.cotuca.unicamp.br;Initial Catalog=BD22129";
SqlConnection conexao = new SqlConnection(cadConexao);
SqlCommand comSql;
SqlDataReader drDados;

string comando =
    "Select customerID, orderID, count(unitPrice) as Items, "+
    "sum(unitPrice * Quantity) as total from Orders "+
    "Inner Join [Orders Details] "+
    "On Orders.orderID = [Orders Details].orderID "+
    "Where Orders.customerID = '" +txtId.Text+"' "+
    "Group By Orders.orderID, Orders.customerID";
comSql = new SqlCommand(comando, conexao);

```

```

conexao.Open();
drDados = comSql.ExecuteReader(CommandBehavior.CloseConnection);
dgv.DataSource = drDados;
dgv.DataBind();
drDados.Close();

```

O comando de consulta (a query Select) foi montado em tempo de execução, sem se verificar o que o usuário digitou no campo txtId, cujo conteúdo é, simplesmente, concatenado a uma string que formará um comando Select nesse banco de dados.

Esses comandos fazem a conexão a um banco de dados e realizam uma query nesse banco, usando como comando Select a string que foi montada logo antes de se instanciar o SqlCommand. O resultado dessa consulta é associado e armazenado em um dataGridView, chamado dgv.

Neste exemplo, um usuário pode tentar burlar o comando SQL. Frequentemente, o primeiro objetivo de tal ataque é receber uma mensagem de erro. Se o erro não é tratado adequadamente e a informação de baixo nível é exposta ao atacante, essa informação poderá ser usada para lançar um ataque mais sofisticado.

Por exemplo, imagine o que aconteceria se o usuário digitasse o seguinte texto no textBox:

ALFKI' or '1='1

Agora considere o comando SQL completo que isto cria:

```

Select customerID, orderID, count(unitPrice) as Items,
sum(unitPrice * Quantity) as total from Orders
Inner Join [Orders Details]
On Orders.orderID = [Orders Details].orderID
Where Orders.customerID = 'ALFKI' or '1='1'
Group By Orders.orderID, Orders.customerID

```

Este comando retorna todos os registros de pedidos da tabela Orders.

Mesmo se o pedido não tenha sido criado pelo cliente ALFKI, ainda é verdadeiro que $1 = 1$ para cada linha de dados.

Assim, ao invés de vermos as informações específicas do cliente atual, todas as informações são expostas ao atacante, como mostrado na figura ao lado.

Se a informação apresentada na tela é sensível, tais como números de CPF ou de Seguridade Social, datas de nascimento, números de cartões de crédito, isto pode se tornar um enorme problema!

CustomerID	OrderID	Items	Total
VINET	10248	3	440.0000
TOMSP	10249	2	1863.4000
HANAR	10250	3	1813.0000
VICTE	10251	3	670.8000
SUPRD	10252	3	3730.0000
HANAR	10253	3	1444.8000
CHOPS	10254	3	625.2000
RICSU	10255	4	2490.5000
WELLI	10256	2	517.8000
HILAA	10257	3	1119.9000
FRNSH	10258	3	2018.6000

De fato, comandos simples de SQL Injection exatamente como este são frequentemente a origem de problemas que afetam grandes companhias de comércio eletrônico. Frequentemente, a vulnerabilidade não ocorre em um textBox mas aparece na query string de uma página Web (que pode ser usada para passar valores de bancos de dados tais como um ID único).

Ataques mais sofisticados são possíveis. Por exemplo, o usuário malicioso pode simplesmente comentar o resto de seu comando SQL ao adicionar dois hífens (--). Este ataque é específico de SQL Server, mas outras artimanhas são possíveis em MySQL com o uso do símbolo " # " e em Oracle com " ; ". De forma alternativa, o atacante pode usar um comando de lotes para executar um comando SQL arbitrário. Com o provedor SQL Server, o atacante simplesmente precisa fornecer um ponto-e-vírgula seguido por um novo comando. Essa artimanha permite ao usuário apagar os conteúdos de outra tabela, ou mesmo usar a stored procedure de sistema xp_cmdshell para executar um programa qualquer na linha de comando do SQL Server.

Eis como o usuário precisaria digitar o textbox para um ataque de SQL Injection mais sofisticado, para apagar todas os registros da tabela Customer (de clientes):

ALFKI'; delete * from customers

O comando SQL resultante seria, na verdade, dois comandos, com um comentário ao final:

```
Select customerID, orderID, count(unitPrice) as Items,
sum(unitPrice * Quantity) as total from Orders
Inner Join [Orders Details]
On Orders.orderID = [Orders Details].orderID
Where Orders.customerID = 'ALFKI';
delete * from customers--Group By Orders.orderID, Orders.customerID
```

Portanto, como você pode se defender contra ataques de SQL Injection? Existem algumas diretrizes que você pode sempre ter em mente:

1. Usar a propriedade textBox.MaxLength para prevenir a digitação de dados muito longos se eles não são necessários. Isto reduz a chance de um grande bloco de script SQL ser copiado onde ele não pertença.
2. Use máscaras de formato em campos de digitação e os controles de validação do framework que usa para programar a aplicação, de forma a impedir dados obviamente incorretos, tais como texto, espaço ou caracteres especiais em campos numéricos.
3. Restrinja as informações apresentadas por suas mensagens de erro. Se você captura uma exceção de banco de dados, você deve relatar apenas uma mensagem genérica tal como "Erro na origem de dados" ao invés de apresentar a informação completa da função Error_Message(), como fizemos acima. Essa mensagem completa pode fornecer mais informações sobre as vulnerabilidades do sistema.
4. A mais importante de todas as diretrizes é remover caracteres especiais. Por exemplo, você pode converter **todas as apóstrofes** em duas apóstrofes assegurando, dessa forma, que eles não sejam confundidos com os delimitadores em seu comando SQL:

```
string comando =
"Select customerID, orderID, count(unitPrice) as Items, "+
"sum(unitPrice * Quantity) as total from Orders "+
"Inner Join [Orders Details] "+
"On Orders.orderID = [Orders Details].orderID "+
"Where Orders.customerID = '" +
txtID.Text.Replace("'''", "''") + "' " +
"Group By Orders.orderID, Orders.customerID";
```

Obviamente, essa diretriz introduz dores de cabeça se os valores de seu texto realmente precisarão conter apóstrofes. Em alguns casos, ataques de SQL Injection ainda serão possíveis. Substituir apóstrofes impede que um usuário malicioso feche um valor string prematuramente. No entanto, se você está criando dinamicamente um comando SQL que inclui valores numéricos na cláusula where, um ataque de SQL Injection necessitará apenas de um único espaço em branco. Essa vulnerabilidade é frequentemente (e perigosamente) ignorada.

5. Uma abordagem ainda melhor é usar um comando parametrizado ou uma stored procedure que realiza sua própria saída do script asp.net e não é suscetível a ataques de SQL injection. A seguir estudaremos essa abordagem.

Usando comandos parametrizados

Um comando parametrizado é, simplesmente, um comando que usa trechos reservados e substituíveis no texto SQL. Os trechos reservados (ou parâmetros) indicam valores fornecidos dinamicamente, que serão enviados ao servidor de banco de dados através da coleção Parameters do objeto SqlCommand.

Por exemplo, observe o comando abaixo:

```
Select * from Customers where customerID = 'ALFKI'
```

Na programação em C# ele se tornaria algo semelhante a este:

```
Select * from Customers where customerID = @custID
```

`@custID` é o trecho reservado e usado para armazenar um valor fornecido pelo usuário. Ele será indicado num objeto SqlCommand como um parâmetro, que será substituído ao ser enviado ao servidor de Bancos de Dados. Nessa substituição, os apóstrofes a mais são excluídos **automaticamente**; comentários e pontos-e-vírgulas passam a fazer parte da string que será comparada e não gerarão novos comandos SQL embutidos no Select acima nem deixarão comentados as cláusulas que vierem após "- -". Ou seja, é feita uma "limpeza" ou "sanitização" do comando Select, impedindo seu uso para SQL Injection.

Assim, primeiramente criamos um SqlCommand contendo a string com o comando Select da consulta desejada. Essa string pode ser colocada em uma variável ou ser colocada diretamente como um argumento do construtor, caso em que vemos no exemplo abaixo:

```
SqlCommand meuComando = new SqlCommand(
    "Select customerID, orderID, count(unitPrice) as Items, "+
    "sum(unitPrice * Quantity) as total from Orders "+
    "Inner Join [Orders Details] "+
    "On Orders.orderID = [Orders Details].orderID "+
    "Where Orders.customerID = @custID "+
    "Group By Orders.orderID, Orders.customerID" ,
    conexao);
```

Em seguida, devemos configurar o objeto SqlCommand (chamado de meuComando nesse exemplo) para que à sua coleção de parâmetros seja adicionado um novo parâmetro, já fornecendo seu valor:

```
meuComando.Parameters.AddWithValue("@custId", txtId.Text);
```

Após isso, abriremos a conexão, executaremos o SqlCommand e recuperaremos o result set do comando Select, armazenando-o em um SqlDataReader.

O SqlDataReader será o DataSource de um GridView, de forma que os registros trazidos pela execução com SqlCommand serão apresentados em formato tabular na página web ou formulário Desktop que estamos criando:

```
conexao.Open();
SqlDataReader drLeitor = meuComando.ExecuteReader();
dgv.DataSource = drLeitor;
dgv.DataBind();
drLeitor.Close();
conexao.Close();
```

O comando executado no servidor de banco de dados será:

```
Select customerID, orderID, count(unitPrice) as Items,
sum(unitPrice * Quantity) as total from Orders
Inner Join [Orders Details]
On Orders.orderID = [Orders Details].orderID
Where Orders.customerID = [ALFKI' OR '1'='1']
Group By Orders.orderID, Orders.customerID
```

Se um usuário tentar realizar um ataque com SQL Injection contra esta versão revisada da página, nenhum registro será retornado. Isso ocorre porque nenhum pedido contém um customerId igual à string ALFKI' OR '1'='1, que é como a cadeia digitada será tratada pelo SQL Server. Esse é exatamente o comportamento que desejamos para evitar os ataques de SQL Injection.

Usando Stored Procedures

Os comandos parametrizados são apenas um pequeno passo prévio aos comandos que chamam stored procedures. A seguir temos o código SQL necessário para criar um stored procedure para inserir um único registro na tabela Employees, do banco de dados de exemplo Northwind, que é um banco de dados fornecido pela Microsoft para estudo:

```
CREATE PROCEDURE pInsereFuncionario
    @titulo as varchar(25),
    @sobrenome as varchar(20),
    @nome as varchar(10),
    @idFuncionario as int OUTPUT
AS
BEGIN
    INSERT INTO Employees (TitleOfCourtesy, LastName, FirstName, HireDate)
    VALUES (@titulo, @sobrenome, @nome, GetDate())          -- parâmetros de entrada
    SET @idFuncionario = @@IDENTITY                         -- parâmetro de saída
END
```

Esse stored procedure recebe três parâmetros para o título de cortesia, sobrenome e nome da pessoa. Ele retorna o id do novo registro através do **parâmetro de saída** chamado @idFuncionario, que é recuperado após o comando INSERT através da função @@IDENTITY. Este é um exemplo de uma simples tarefa que um stored procedure torna muito mais fácil. Sem o uso da stored procedure, seria bastante complicado determinar, no programa escrito em C#, o valor da chave primária autoincremento (@EmployeeID) gerada automaticamente pelo servidor para essa tabela, logo após ter-se incluído um novo registro.

Esse código deve ser feito no banco de dados.

Em seguida, deveremos voltar ao código C# e criar o objeto de conexão ao banco de dados. Ao usuário indicado na cadeia de conexão deverá ser concedido (GRANT) o direito de execução do

stored procedure pInsereFuncionario. Esse usuário não terá que ter permissão de Inserção na tabela, apenas o de executar o procedimento armazenado.

```
SqlConnection conexao= null;
try
{
    conexao = new SqlConnection(
        "Data Source=Regulus.cotuca.unicamp.br;Initial Catalog=BD22129");
}
```

Após isso, instanciamos um objeto SqlCommand para encapsular a chamada ao stored procedure. Esse comando terá os mesmos três parâmetros como entradas e terá um quarto parâmetro de saída que retornará o valor do campo EmployeeID referente ao último registro inserido na tabela.

```
SqlCommand sqlCom = new SqlCommand("pInsereFuncionario", conexao );
sqlCom.CommandType = CommandType.StoredProcedure;
```

Agora precisamos adicionar os parâmetros da stored procedure à coleção Parameters do SqlCommand. Quando você fizer isso, precisará especificar **explicitamente** o tipo de dados exato e o comprimento do parâmetro de forma que ele combine com os detalhes de descrição dos campos do banco de dados. Abaixo temos como fazer isso para o primeiro parâmetro:

```
sqlCom.Parameters.Add( new SqlParameter("@titulo", SqlDbType.VarChar, 25) );
sqlCom.Parameters["@titulo"].Value = txtTitulo.Text;
```

A primeira linha cria um novo objeto SqlParameter. Ele configura o nome do parâmetro, seu tipo (usando a enumeração SqlDbType) e seu tamanho (como um número de caracteres máximo) no construtor. Em seguida esse novo parâmetro é adicionado à coleção Parameters do SqlCommand. O segundo comando atribui o valor do parâmetro, que será enviado à stored procedure quando o sqlCommand for executado. Agora podemos adicionar os dois próximos parâmetros de uma forma semelhante:

```
sqlCom.Parameters.Add( new SqlParameter("@sobrenome", SqlDbType.VarChar, 20) );
sqlCom.Parameters["@sobrenome"].Value = txtSobrenome.Text;
sqlCom.Parameters.Add( new SqlParameter("@nome", SqlDbType.VarChar, 10) );
sqlCom.Parameters["@nome"].Value = txtNome.Text;
```

O último parâmetro é um parâmetro de saída, que permite à stored procedure retornar informação ao seu código em C#. Embora esse objeto Parameter seja criado da mesma maneira, você deve garantir que ele seja especificado como um parâmetro de saída ajustando sua propriedade **Direction** para **Output**. Você não precisa fornecer um valor para esse parâmetro:

```
sqlCom.Parameters.Add( new SqlParameter("@idFuncionario", SqlDbType.Int, 4) );
sqlCom.Parameters["@idFuncionario"].Direction = ParameterDirection.Output;
```

Por fim, abrimos a conexão e executamos o SqlCommand com o método ExecuteNonQuery(). Quando o comando terminar, você poderá ler o valor do parâmetro de saída, como visto abaixo:

```
conexao.Open();
int numRegAfetados = sqlCom.ExecuteNonQuery();
lblMensagem.Text = $"Inseridos {numRegAfetados} registros.");
// Obtém o ID de funcionário recém gerado no servidor de Banco de dados
int IdFunc = (int)sqlCom.Parameters["@idFuncionario"].Value;
lblMensagem.Text += " Novo ID de funcionário:" + idFunc.ToString();
}
```

```

catch (Exception ex)
{
    lblmsg.Text = ex.Message;
    enviarEmailDeErroAoAdministrador(ex.Message);
}
finally
{
    conexao.Close();
}

```

Adicionando Parâmetros com Tipos de dados Implícitos

Um atalho interessante é o método `AddWithValue()` da coleção `Parameters`. Esse método recebe o nome do parâmetro e o valor mas nenhuma informação de tipo de dados. Ao invés disso, ele **infere** o tipo de dados através dos valores fornecidos (obviamente, isso funciona com parâmetros de entrada mas não com parâmetros de saída). Se você não precisa explicitamente escolher um tipo de dados não-padrão, poderá simplificar seu código usando essa abordagem menos restritiva.

Abaixo temos um exemplo onde, usando um possível banco de dados de questionários de alunos, podemos usar a estratégia discutida acima para acessar os dados dos alunos, por exemplo. Criamos um stored Procedure no banco de dados, com um parâmetro ou mais:

```

CREATE PROCEDURE pConsultaAluno
    @RA as varchar
AS
BEGIN
    SELECT * from aluno where RA = @RA
END
GO

```

```

SqlConnection conexao= null;
try
{
    conexao = new SqlConnection(
        "Data Source=Regulus.cotuca.unicamp.br;Initial Catalog=BD22129");
    SqlCommand sqlCom = new SqlCommand("pConsultaAluno", conexao );
    sqlCom.CommandType = CommandType.StoredProcedure;

    sqlCom.Parameters.AddWithValue("@RA", txtRA.Text);

    conexao.Open();
    SqlDataReader drAluno = sqlCom.ExecuteReader();
    dgv.DataSource = drAluno;
    dgv.DataBind();
}
catch (Exception ex)
{
    lblmsg.Text = ex.Message; // isto não deveria ser feito!!!
}
finally
{
    conexao.Close();
}

```

Assumindo que `txtRA.Text` é uma string com 5 caracteres, o método `AddWithValue` criará um objeto `SqlParameter` com `Size` de 5 (caracteres) e um `SqlDbType` de `NVarChar` (`varchar Unicode`, com 2 bytes por caracter). O Banco de dados converterá esse dado conforme necessário, desde

que você não esteja tentando colocar dados em um campo de tamanho menor ou com um tipo de dados totalmente diferente.

Observação: Existe um problema - campos que permitem valores nulos. Se você desejar passar um valor null para um stored procedure, de forma que ela possa usar o valor null para armazenar em um campo, por exemplo, você não poderá usar a palavra reservada **null** do C#, que é a referência para um endereço nulo. Infelizmente, você também não pode usar um tipo de dado anulável (tal como int?), porque a classe SqlParameter não suporta tipos de dados anuláveis. Para indicar o conteúdo nulo em um campo, você deve passar a constante do .Net **DBNull.Value** como o valor do parâmetro.

Vídeo adicional: [Como usar Stored Procedure do SQL Server numa aplicação C#](#)

Fontes de consulta adicionais:

<https://learn.microsoft.com/pt-br/training/modules/create-stored-procedures-table-valued-functions/>
<https://learn.microsoft.com/pt-br/sql/t-sql/language-elements/control-of-flow?view=sql-server-ver16>
https://www.w3schools.com/sql/sql_stored_procedures.asp#:~:text=SQL%20Stored%20Procedures%20for%20SQL%20Server&text=A%20stored%20procedure%20is%20a,call%20it%20to%20execute%20it.
<https://www.devmedia.com.br/introducao-aos-stored-procedures-no-sql-server/7904>
<https://www.navicat.com/en/company/aboutus/blog/2053-are-stored-procedures-an-outdated-tool.html>
<https://www.sqlservertutorial.net/sql-server-stored-procedures/>

7.7. Curores e Transações

Cursor é o nome dado para o resultado de uma consulta a um banco de dados (um Select) que possa ser processado por um Stored Procedure como se fosse um arquivo cujos dados são percorridos (lidos um a um), analisados e, eventualmente, participem de cálculos e atualizações.

Portanto, um cursor é como um arquivo que podemos ler sequencialmente, atualizar dados, apagar registros, etc. Esse arquivo é temporário, pois existe somente na memória enquanto o cursor está sendo processado.

Curores podem ser muito úteis em situações que exijam o processamento de dados em sequência, por um algoritmo que exija o uso de If, While e outros comandos para processar tais dados. Ou seja, quando não é possível usar somente comandos mais simples como Select, Insert, Update ou Delete, e haja necessidade de um algoritmo mais complexo para implementar uma ou mais regras de negócio, podemos usar curores.

Os curores são criados e processados no servidor de banco de dados, já que são criados em stored procedures geralmente. Criá-los e processá-los em stored procedure tem algumas vantagens:

1. O result set não trafega pela rede até o cliente para ser processado
2. O código de processamento dos dados retornados (Regra de Negócio) é armazenado no próprio servidor e não na aplicação cliente; se houver alguma alteração no código, ela estará prontamente disponível para todas as aplicações clientes, que não precisarão ser recompiladas nem redistribuídas;
3. Atualizações dos dados são feitas no próprio servidor e não na aplicação cliente, de forma que diminui o tráfego de rede

Declaração de cursor:

```
declare <nome do cursor> cursor for
      <comando select>
```

Exemplo:

```
declare cCestaCompra Cursor for
      Select
```

```

CC.idItemCompra, CC.idCliente, CC.dataCompra,
CC.quantidadeVendida, P.idProduto, P.Estoque, P.ValorUnitario
from
Loja.CestaCompra CC Inner JOIN Loja.Produto P
on P.idProduto = CC.idProduto
where
idCliente = @pIdCliente
order by
dataCompra

```

@pIdCliente seria um parâmetro passado para o Stored Procedure onde esse cursor está sendo declarado. Como se vê acima, embora sejam declarados no comando **DECLARE**, cursores não têm @ no início de seu nome.

Como um exemplo de aplicação de um cursor, imagine o processamento de dados nas seguintes situações:

Sistema de Controle Acadêmico

1. Uma escola possui um banco de dados com uma tabela de alunos, de disciplinas do currículo e uma tabela que relaciona o desempenho de cada aluno em cada disciplina em que está matriculado.
2. No final do período letivo (semestre ou ano letivos) o sistema deverá executar uma stored procedure que, para cada aluno matriculado, verifique sua situação final em cada disciplina e sua situação geral no semestre;
3. Cria-se um cursor baseado na tabela de alunos, contendo apenas os alunos de uma determinada classe cujo encerramento de semestre está sendo feito;
4. Esse cursor será percorrido sequencialmente, aluno a aluno (registro a registro);
5. Para cada registro de aluno, será criado um outro cursor, contendo os registros desse aluno nas disciplinas em que está matriculado. Nesse registro teremos informações como o RA do aluno, o código da disciplina, a média final, a nota de recuperação, a frequência nas aulas e um campo que indica a situação do aluno no momento (cursando, trancado, desistente, etc.)
6. Esse segundo cursor será processado sequencialmente e, para as informações de média final e frequência, será aplicado o algoritmo que define a nova situação do aluno (aprovado, em recuperação, reprovado, em conselho, etc), se esse aluno não trancou a matrícula na disciplina ou foi considerado desistente.
7. Após se definir a situação do aluno na disciplina atualmente processada, é feito um Update no registro do aluno nessa tabela e nessa disciplina, atualizando a situação do mesmo. É contado, também, a quantidade de cada tipo de situações;
8. Em seguida, quando se terminar de processar esse cursor, a quantidade de cada tipo de situações definirá a situação geral do aluno no período letivo (aprovado, reprovado, em dependência, etc.)
9. Esse cursor de disciplinas do aluno será processado até seu final; quando isso acontecer, ele será fechado e removido da memória, para liberar recursos de memória e processador do computador servidor de banco de dados;
10. Parte-se então para o processamento do próximo registro do cursor de alunos; quando este terminar, esse cursor será fechado e removido da memória, para liberar recursos de memória e processador do computador servidor de banco de dados;

Sistema de vendas online

1. A sacola ou cesta do cliente da loja fica armazenada no banco de dados, na tabela Cesta; O cliente vai comprando e cada item (produto) comprado é armazenado em um registro nessa tabela;
2. Cada registro contém o id do cliente, a data de compra, id do produto, quantidade comprada;

3. Quando o cliente da loja decide finalizar a compra, os registros desse cliente armazenados na Cesta são trazidos, via Select, para o stored procedure e guardados em um cursor;
4. O cursor é percorrido sequencialmente (lido) e cada registro da cesta é processado.
5. Esse processamento envolve verificar se há estoque desse produto na tabela de estoque para atender a quantidade comprada, calcular o preço de cada item comprado (preço unitário x quantidade comprada), somar esse preço no total a pagar, dar baixa dessa quantidade no estoque;
6. Ao final do processamento do cursor, deve-se processar a forma de pagamento e gerar tantos registros de parcela a pagar quanto for o número de parcelas em que o cliente dividiu o pagamento; também deve-se verificar se o meio de pagamento e preparar os registros necessários para pagamento via PIX, via Débito, via Crédito ou outros.
7. Quando terminar o processamento do cursor, este será fechado e removido da memória, para liberar recursos de memória e processador do computador servidor de banco de dados;

Ainda não vimos como um cursor é processado, mas podemos concluir que são usados em situações que exigem algoritmos mais complexos e que não são facilmente implementados somente com comandos Insert, Update, Delete e Select. Cursores, portanto, são associados a processos das regras de negócio de um sistema de banco de dados, principalmente no processamento registro a registro em momentos de atualização e consolidação de dados.

Processamento de um Cursor

O processamento de um cursor envolve 4 fases: abertura, percurso, fechamento e desalocação.

Após [declarar um cursor](#), para ativá-lo é necessário abri-lo. Para abrir um cursor, usamos o comando OPEN nomeDoCursor, como vemos abaixo:

```
OPEN cCestaCompra
```

Quando isso é feito, o comando **Select** associado do cursor é executado no servidor de banco de dados, e os dados resultantes dessa consulta são trazidos para o lote de comandos onde o cursor foi declarado e aberto, ficando disponíveis para percurso e utilização pelo algoritmo de tratamento desses dados.

Após percorrer o cursor totalmente, devemos fechá-lo e desaloca-lo, usando os comandos abaixo:

```
CLOSE cCestaCompra  
DEALLOCATE cCestaCompra
```

CLOSE libera os recursos usados para manter o conjunto do cursor e também liberta quaisquer bloqueios que tenham sido colocados nos registros processados. Bloqueios podem ser colocados para impedir que um registro em processamento no cursor seja alterado por algum outro Stored Procedure ou aplicação externa ao banco de dados que esteja sendo acessado simultaneamente.

DEALLOCATE remove da memória o cursor e o deixa disponível para nova abertura, caso seja necessária.

Para percorrer o cursor, temos que ler cada linha de dados do resultado retornado pelo Select e processá-la. Para ler uma linha de dados, usamos o comando FETCH. Para ler várias linhas em seguida, teremos de usar um comando de repetição While, que fará a leitura sequencial de cada linha enquanto o conjunto de dados não terminar, da mesma forma que se lê um arquivo de dados até que este termine, em outras linguagens de programação.

```
Fetch nomeDoCursor Into variavel1, variavel2, ...
```

Para saber se um cursor foi percorrido totalmente, usamos a função @@Fetch_Status. Enquanto seu resultado for **igual a zero**, ainda restam linhas para serem lidas do cursor.

Como @@FETCH_STATUS é global para todos os cursores em uma conexão, use-o com cuidado. Depois que uma instrução FETCH é executada, o teste para @@FETCH_STATUS deve ocorrer antes que qualquer outra instrução FETCH seja executada com relação a outro cursor. @@FETCH_STATUS é indefinido antes de ocorrer qualquer busca na conexão.

Por exemplo, um usuário executa uma instrução FETCH a partir de um cursor e, depois, chama um procedimento armazenado que abre e processa os resultados de outro cursor. Quando o controle retorna desse procedimento armazenado chamado, @@FETCH_STATUS reflete o último FETCH executado no procedimento armazenado, não a instrução FETCH executada antes de chamar o procedimento armazenado.

Abaixo temos um exemplo usando um cursor para atualizar situações de alunos de uma classe:

```
--      exec spAtualiza_Situacao_Classe 45, @tot_apr output, @tot_ret output
--      @tot_rec ouput

create procedure spAtualiza_Situacao_Classe
    @ID_Classe int,                      -- informa a classe desejada para atualização
    @quantos_apr int output,              -- devolve o número de alunos aprovados
    @quantos_ret int output,              -- devolve o número de alunos reidos
    @quantos_rec int output,              -- devolve o número de alunos em recuperação
as
begin
    Declare
        @RA_Aluno char(5),                -- Declaração das variáveis locais
        @Situacao_Geral int,              -- RA lido do cursor cAluno
        @Nota float,                     -- situação atual do aluno lido
        @Situacao_Disciplina int,        -- Situacao atual do aluno na disciplina lida
        @Cod_Disc char(5),               -- Código da disciplina sendo processada
        @Disc_Ret int,
        @Disc_Rec int,
        @Disc_Apr int

    Declare cAluno Cursor for
        Select RA, Situacao_Geral
        From Aluno
        where ID_Classe = @ID_Classe

    Open cAluno -- abre, disponibiliza o cursor, e busca os registros
    Fetch cAluno into @RA_Aluno, @Situacao_Geral -- lê o 1º registro
    while @@fetch_status = 0 -- se fetch_status = 0 ---> o cursor terminou
    begin
        If @Situacao_Geral in (1,6,7,8,9) -- aluno está cursando
        begin
            declare cDisciplina cursor for
                SelectCodigo_Disciplina, Nota, -- busca todas as disciplinas
                    Situacao_Geral           -- realizadas pelo aluno cujo RA
                from Disciplina             -- é igual ao valor armazenado em
                where RA = @RA_Aluno       -- @RA_Aluno
            Set @Disc_Ret = 0           -- zera os contadores de situação nas
            Set @Disc_Rec = 0           -- disciplinas para depois verificar a
            Set @Disc_Apr = 0           -- situação geral final do aluno

            Open cDisciplina
            Fetch cDisciplina into @Cod_Disc, @Nota, @Situacao_Disciplina
            while @@fetch_Status = 0 -- não acabou cDisciplina
            begin
                if @Situacao_Disciplina in (1,6,7,8,9,10) - ativo na disciplina

```

```

begin
if @Nota < 3
begin
    set @Disc_ret += 1           -- mais uma disciplina retilda
    set @Situacao_Disciplina = 7 -- código de retenção
end
else
    if @Nota < 5
    begin
        set @Disc_rec += 1       -- mais uma disciplina em recuperação
        set @Situacao_Disciplina = 6 -- código de recuperação
    end
    else
        begin
            set @Disc_Apr += 1     -- mais uma disciplina aprovada
            set @Situacao_Disciplina = 9 -- código de aprovação
        end
end

Update Disciplina
    set Situacao_Disciplina = @Situacao_Disciplina
    where RA = @RA_Aluno and Codigo_Disciplina = @Cod_Disc
End

Fetch cDisciplina into @Nota, @Situacao_Disciplina
end -- while não acabou CDisciplina
Close cDisciplina
Deallocate cDisciplina

if @Disc_ret <> 0           -- o aluno está retildo em alguma disciplina
begin
    set @Situacao_Geral = 7   -- retildo
    set @quantos_ret += 1     -- mais um aluno retildo
end
else
    if @Disc_Rec <> 0        -- o aluno está em recuperação em alguma disciplina
    begin
        set @Situacao_Geral = 6   -- em recuperação
        set @quantos_rec += 1     -- mais um aluno em recuperação
    end
    else
        begin
            set @Situacao_Geral = 9   -- aprovado
            set @quantos_rec += 1     -- mais um aluno aprovado
        end

Update Aluno
    Set Situacao_Geral = @Situacao_Geral
    Where RA = @RA_Aluno
end
Fetch cAluno into @RA_Aluno, @Situacao_Geral -- lê próximo registro
end
Close cAluno          -- fecha o cursor
Deallocate cAluno    -- retirar o cursor da memória
End

```

Um outro exemplo segue abaixo:

```

create procedure spEfetivaVenda @pIdCliente int, @pQtasParcelas int
as
begin
    Declare @IdItemCompra int,           -- variáveis usadas para

```

```

@IdCliente int,          -- armazenar os valores lidos
@DataCompra Datetime,    -- do cursor
@QuantVendida int,
@IdProduto int,
@QuantEstoque int,
@ValorUnitario Money,
@DataVenda DateTime,
@idVenda int,
@ValorTotal Money,
@numParcela int,
@valorParcela Money,
@DataVencimento DateTime,
@erro int

Declare cCestaCompra Cursor for
Select
    CC.idItemCompra, CC.idCliente, CC.dataCompra,
    CC.quantidadeVendida, P.idProduto, P.Estoque, P.ValorUnitario
from
    Loja.CestaCompra CC Inner JOIN Loja.Produto P
        on P.idProduto = CC.idProduto
where
    idCliente = @pIdCliente
order by
    dataCompra

set @DataVenda = GetDate() -- data atual, para, junto ao Id_Cliente,
                           -- identificarmos esta venda, pois o IdVenda dessa
                           -- tabela não é devolvido após o Insert

Begin Tran
Insert
    into Loja.Venda (IdCliente, DataVenda, QtasParcelas)
    values (@pIdCliente, @DataVenda, @pQtasParcelas)

    -- busca o IdVenda recém-gravado

Select @idVenda = IdVenda
    from Loja.Venda
    where IdCliente = @pIdCliente and           -- usamos estas variáveis para
        DataVenda = @DataVenda                 -- buscar o IdVenda recém-gravado

Open cCestaCompra -- abre o cursor e traz os dados resultantes do where
Fetch cCestaCompra      -- lê um registro e armazena os campos nas variáveis
    into @IdItemCompra, @IdCliente, @DataCompra, @QuantVendida, @IdProduto,
        @QuantEstoque, @ValorUnitario
set @ValorTotal = 0
set @NumItem = 0
set @erro = 0
while @@Fetch_Status = 0      -- @@Fetch_Status é função que devolve o resultado
begin                      -- do Fetch anterior; 0 indica que foi lido um registro
    -- processa o registro atual
    -- inclui um registro na tabela ItemVenda desta venda, com o produto atual
    set @NumItem += 1
    if @QuantVendida <= @QuantEstoque      -- estoque atende o pedido totalmente
    begin
        Insert into Loja.ItemVenda (IdVenda, NumItem, DataLiberacao,
                                       idProduto, quantVendida, valorUnitario) )
        values (@@Fetch_Status, @NumItem, @DataVenda,
                @idProduto, @quantVendida, @valorUnitario)
        Update Loja.Produto set Estoque = @QuantEstoque - @QuantVendida
        Where idProduto = @idProduto
    end
end

```

```

        Set @ValorTotal += @quantVendida * @valorUnitario
    end
    else -- lote de estoque atual não atende totalmente o pedido
begin
    Rollback Tran
    set @erro = 1
    Break
end
Fetch cCestaCompra -- lê novo registro e armazena os campos nas variáveis
into @IdItemCompra, @IdCliente, @DataCompra, @QuantVendida, @IdProduto,
      @QuantEstoque, @ValorUnitario
End
close cCestaCompra -- fecha o cursor
Deallocate cCestaCompra
if @erro = 0 -- não houve erros
begin
    set @ValorParcela = @ValorTotal / @QtasParcelas
    set @NumParcela = 1
    set @DataVencimento = @DataVenda
    while @NumParcela <= @QtasParcelas
begin
    Insert into Loja.Parcela
        (IdVenda, QualParcela, DataVencimento, ValorPrevisto)
        values (@IdVenda, @NumParcela, @DataVencimento, @ValorParcela)
    set @Num_Parcela += 1
    set @DataVencimento += 30 -- 30 dias depois vence outra parcela
end
Delete from Loja.CestaCompra Where idCliente = @pIdCliente
Commit Tran
End
End

```

<https://www.devmedia.com.br/cursos-no-sqlserver/5590>

8. Programação de Aplicações em Camadas

8.1. Arquitetura de camadas para acesso a bancos de dados

Vamos a um pouco de história, para fornecer-lhe uma base de comparação entre as diversas tecnologias que tem sido usadas para processamento de bancos de dados.

Arquitetura de uma camada

No início da utilização dos computadores para processar grandes lotes de dados, não havia uma separação bastante distinta entre o acesso aos dados e seu processamento. Em outras palavras, um programa precisava acessar os dados diretamente e, para isso, precisava conhecer totalmente a estrutura de armazenamento dos arquivos.

Esse “conhecimento da estrutura” tinha que ser reproduzido em cada programa que acessava os dados. Assim, era difícil separar funcionalmente o acesso aos dados do código do algoritmo de processamento.

Em geral, esse tipo de programa é escrito em linguagens de programação de terceira geração, como COBOL e Clipper. Mesmo que a linguagem Clipper use conceitos de bancos de dados relacionais, como tabelas, índices, relacionamentos, os programas nela desenvolvidos usam o que chamaremos de um sistema de arquivos livres.

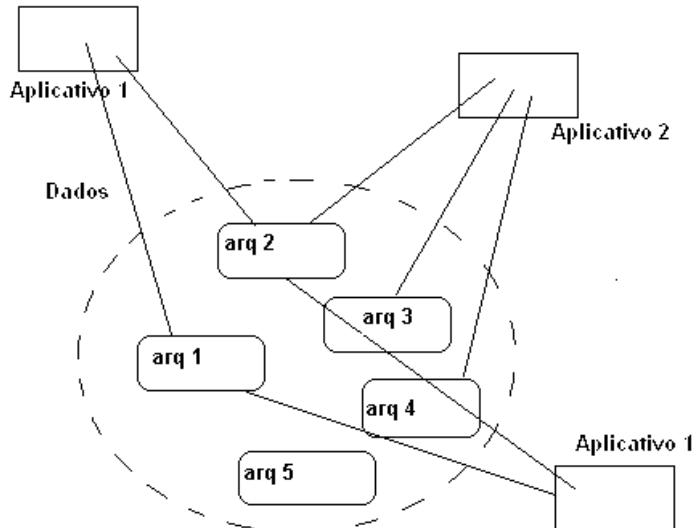
Mesmo que esses arquivos estejam disponíveis em um servidor de arquivos, ou seja, sejam acessíveis através de uma rede local, vários problemas ocorrerão no processamento dos dados quando houverem vários usuários acessando os dados concomitantemente:

- Integridade das informações
- Segurança no acesso
- Desempenho
- Controle de concorrência

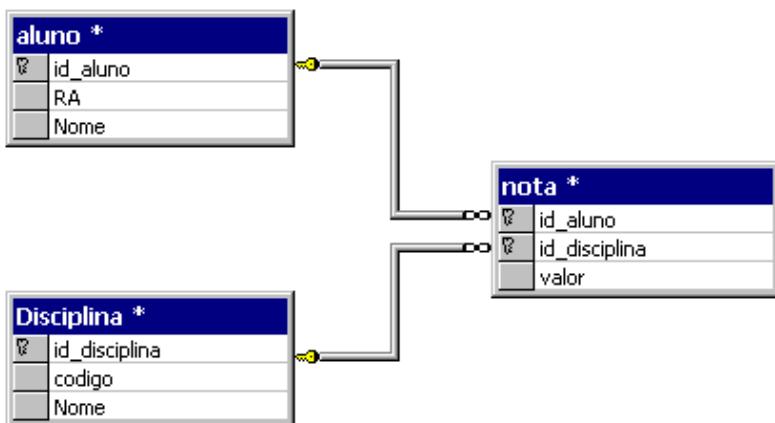
Como toda lógica de acesso aos dados não reside no servidor de arquivos mas, ao contrário, é definida em cada programa que acessa os dados, a integridade das informações deve ser programada em cada nova aplicação que use os dados. Portanto, a redundância de código será um fator de diminuição da qualidade do software. Qualquer mudança nas regras de acesso deverá ser reprogramada em todos os programas afetados. Todas as cópias desses programas deverão ser modificadas.

O controle de integridade pode ser exemplificado pelo uso de relacionamentos entre tabelas fortes e fracas. Quanto uma nota de aluno vai ser incluída na tabela de notas, é necessário que exista a disciplina a que a nota se refere e também o aluno que obteve essa nota. Portanto, como vemos no algoritmo e na figura abaixo, esse relacionamento deveria ser implementado em cada programa que efetue operações de acesso aos dados dessa natureza:

1. Ler RA
2. Se RA existe no arquivo de alunos
 - a. Ler Código da Disciplina
 - b. Se Código da Disciplina existe no arquivo de disciplinas
 - i. Ler Valor da Nota
 - ii. Criar registro no arquivo de notas



- iii. Gravar o registro com as identificações do aluno e da disciplina e o valor da nota



Num sistema de Banco de Dados “de verdade”, o controle do relacionamento e da integridade dos dados seria feito pelo gerenciador de banco de dados, de forma transparente ao programa.

O objetivo é impedir-se que registros órfãos (nota sem disciplina ou sem aluno) apareçam no banco de dados. Quando isso é impedido, o banco é considerado íntegro.

A segurança no acesso deve ser feita para impedir que usuários não-autorizados obtenham acesso a partes sensíveis dos programas e possam, assim, destruir os dados. A dificuldade em implementar esse tipo de controle num sistema de tabelas livres é semelhante ao controle de integridade. Cada novo programa que acessa os dados deverá ter o controle de acesso programado diretamente nele. Qualquer alteração na regras de controle poderá ocasionar a necessidade de manutenção em todos os programas.

Além disso, o diretório de rede onde os dados se encontram devem ser mantidos abertos para leitura e gravação, o que compromete toda a segurança da rede, pois pode-se usar programas como o DBU que vem com Clipper, ou Database Desktop do Delphi para verificar o conteúdo dos arquivos, além de se poder listar, copiar, renomear e apagar arquivos do diretório de dados.

O desempenho fica bastante comprometido num sistema de tabelas livres em rede. Imagine que estamos realizando a verificação da situação final de cada aluno, e gerando um relatório. Para isso, temos algumas regras de negócio (o algoritmo que verifica a situação final). Suponhamos que, para ser considerado aprovado em uma disciplina, o aluno precisa obter nota maior ou igual a 5, e que para ser aprovado na série ele precisa ser aprovado em todas das disciplinas que realiza. Um possível algoritmo básico para isso seria:

```

Enquanto não acabou o arquivo de alunos
    Início
        Ler um registro do arquivo de alunos
        Posicionar o arquivo de notas no primeiro registro desse aluno
        Aprovado ← verdadeiro
        Enquanto (não acabaram os registros desse aluno) e (Aprovado = verdadeiro)
            Início
                Ler um registro do arquivo de notas
                Se nota < 5
                    Aprovado ← False
                Fim
                Se aprovado = False
                    Escreva(RA,'retido')
                Senão
                    Escreva(RA,'Aprovado')
                Fim

```

Imagine agora que estamos num ambiente em rede local. Todos os registros dos arquivos Alunos e Notas serão lidos e trazidos pela rede, aumentando a carga na mesma e diminuindo o desempenho. Muitas vezes, quando ocorre seleção dos dados (por exemplo, queremos apenas alunos de um curso, o que o exemplo acima não trata), mesmo que apenas 20% dos registros sejam de uma determinada classe, todos os outros 80% terão de ser lidos e descartados. Mesmo assim,

eles serão trazidos pela rede, pois o processo de descarte é efetuado pelo programa que está sendo executado em outro computador.

Essa arquitetura de desenvolvimento de programas é chamada de Arquitetura de Uma Camada ou Single-Tier, pois há apenas uma camada de acesso aos dados e nessa mesma camada faz-se o processamento dos dados.

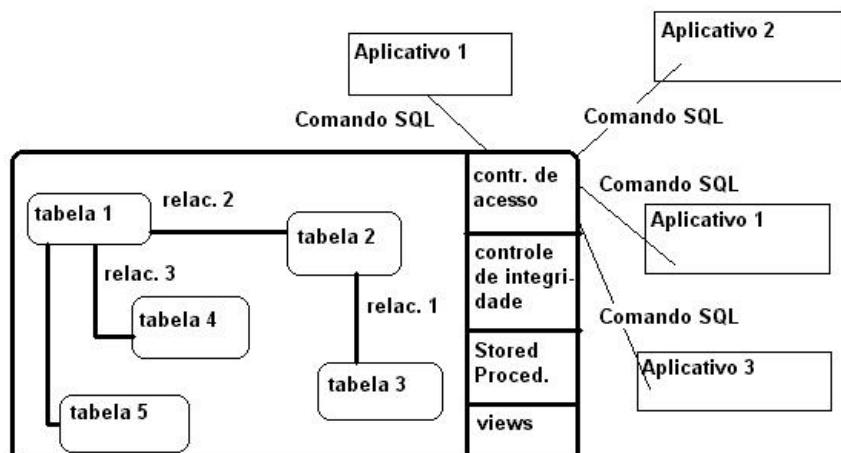
Arquitetura de duas camadas

Num ambiente corporativo (uma empresa), em que os computadores são ligados através de uma rede local, o acesso a um servidor de Banco de Dados Cliente/Servidor, como SQL Server ou Oracle, é muitas vezes bastante vantajoso em relação ao uso de tabelas livres, como Paradox e Dbase. Isso ocorre porque o acesso Cliente/Servidor permite que apenas os registros necessários trafeguem pela rede. Já com tabelas livres, todos os registros de uma tabela trafegam pela rede quando se faz uma consulta (filtragem) aos dados.

Teremos, portanto, um computador destinado a ser o servidor de bancos de dados. Ele executará o sistema gerenciador de bancos de dados (SGBD).

O gerenciador de banco de dados é um programa que é executado em um computador especialmente configurado da rede. Esse computador possivelmente terá mais memória e espaço em disco, além de maior velocidade, do que os computadores que normalmente os usuários utilizam para executar os programas no cotidiano de suas atividades. O programa gerenciador de banco de dados é responsável pela montagem de bancos de dados, sua configuração, descrição dos relacionamentos entre tabelas, chaves primárias e estrangeiras, restrições de integridade, permissões de acesso que cada usuário possui e controle de concorrência, dentre outras tarefas. Apenas ele tem privilégios de acesso aos dados e, portanto qualquer consulta, inclusão, exclusão e alteração de dados deve antes passar pelo servidor para ser analisada e, possivelmente, executada.

Os computadores clientes são aqueles utilizados pelos usuários finais. Nesses computadores temos o que se chama de front-end, ou seja, a aplicação que acessa os dados que estão no servidor. O cliente envia consultas em linguagem SQL ao servidor. Este recebe a consulta (uma cadeia de caracteres), analisa-a e, caso considere-a executável, processa-a, buscando os dados solicitados, ou fazendo inclusões, remoções ou alterações de registros.



No início da utilização dos bancos de dados, geralmente apenas os dados ficavam no servidor. O programa do front-end era responsável pelo envio da cadeia com a consulta e pelo processamento dos dados recebidos.

O servidor desempenhava apenas as funções listadas acima: controle de usuários, controle de relacionamentos, controle da integridade dos dados (um exemplo: quando se inclui um registro de uma entidade fraca, o correspondente na entidade fraca deve existir), chaves e controle de concorrência. Claramente isso é uma boa parte das tarefas que o front-end tradicional teria que desempenhar num sistema de tabelas livres. Essas tarefas, principalmente o controle de concorrência, são bastante difíceis de serem implementadas em cada novo programa que se produz.

No entanto, um computador servidor com grande poder de processamento pode ser utilizado para outras tarefas dentro do próprio banco de dados. Por exemplo, há o que se chama de regras de negócio. Elas são os algoritmos rotineiros que se executam sobre os dados.

Imagine um banco de dados de pessoal: nele, uma das tarefas a serem executadas seria o cálculo da folha de pagamento. Para isso, os dados estão dispostos, depois na normalização, geração do modelo físico e sua implantação no servidor de banco de dados, em algumas tabelas, como a de cadastro de funcionários, a de cadastro de dependentes, a tabela de faixas salariais, tabela de Imposto de Renda, tabelas de cargos, dentre outras.

No esquema original de uso dos bancos de dados, após os dados necessários terem sido acessados pelo servidor e levados ao cliente pela rede, o processamento era feito localmente. As alterações nos registros (como os salários calculados, descontos, etc) eram depois repassados ao servidor e lá gravados novamente nas tabelas.

Pensou-se então na ideia de Procedimentos Armazenados: a lógica de negócios ficaria armazenada também no servidor, em blocos de código SQL chamados de **Stored Procedures**. Esses blocos conteriam os comandos de consulta ao banco de dados e também comandos para processar os dados acessados. Assim, o processamento seria disparado pelo front-end, mas realizado no back-end, que é o nome que passou a ser usado para referenciar os procedimentos executados no servidor.

Várias vantagens existem nessa abordagem. A primeira e mais evidente delas é que os dados deixariam de trafegar pela rede para serem processados, modificados e novamente retornarem ao servidor para regravação. Outra vantagem é que em apenas um único lugar ficaria a lógica de negócio, de maneira que, na eventualidade de ser necessário alterar o algoritmo de alguma tarefa, apenas na stored procedure específica que a implementa é que a alteração seria feita. Nenhum dos front-ends teria de ser modificado, recompilado e redistribuído entre todos os usuários. Assim, todos os programas dos clientes automaticamente executariam o procedimento modificado.

Temos também as **Views** (visões ou visualizações): elas consistem de consultas SQL que deixamos prontas no servidor, pois sabemos que serão bastante usadas nos programas clientes para consultas, emissão de relatório, exibição de listas de dados, etc.

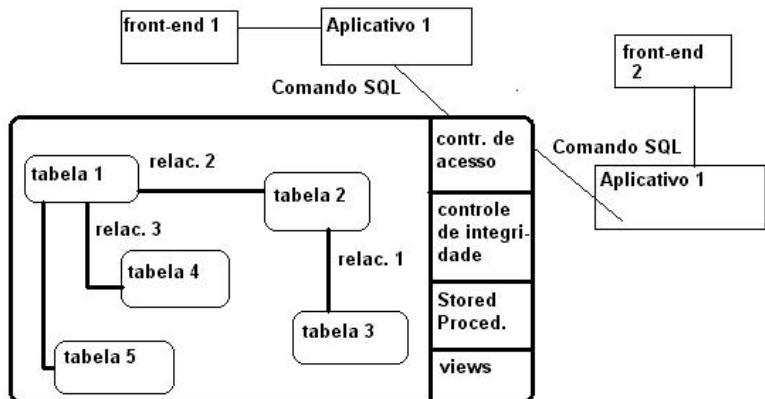
Nessa arquitetura, teremos os programas clientes e o SGBD. Temos, assim, duas camadas para realizar o processamento dos dados. Isso é chamado de Arquitetura de Duas Camadas ou Two-Tier.

Arquitetura de três ou mais camadas

Nem sempre a arquitetura de duas camadas poderá resolver todos os problemas de desempenho e de manutenção dos programas de acesso ao Banco de Dados. Note que muito da lógica de negócio ainda se mantém acoplada ao front-end, que é a interface com o usuário.

Alguns padrões têm surgido e aplicados para desvincular a execução da lógica de negócios tanto do front-end quanto do SGBD.

Assim, temos o que se chama de servidores de aplicação, que são executados a partir da interface com o usuário (front-end) e buscam dados em servidores de bancos de dados (back-end). Processam esses dados e enviam resultados para o front-end e atualizações para o back-end.



Passamos, assim, a ter 3 camadas ou arquitetura Three-Tier. A implementação da lógica de negócios, por estar dissociada do front-end, pode agora ser a mesma usada por diversas formas de acesso aos dados. Por exemplo, podemos numa escola ter as seguintes situações para digitação de notas de alunos:

1. Através do programa de controle acadêmico da secretaria (um programa com todas as funções de acesso e manipulação dos dados). Esse programa seria feito em C#, por exemplo.

2. Através de um aplicativo mobile que permite ao professor digitar notas.
3. Através de um site na Internet, onde uma página segura permite ao professor digitar as notas.

Nesses 3 casos, a lógica de negócio seria a mesma, e portanto, o mesmo programa, que reside em algum computador ligado à Internet teria uma cópia do aplicativo e o deixaria disponível para execução através desses meios de acesso. Note que, no caso de mudança das regras de negócio, o aplicativo só seria modificado em um único lugar.

Modelo em três camadas

Origem: Wikipédia, a enciclopédia livre.

Modelo em três camadas (3-Tier), derivado do modelo '*n*' camadas, recebe esta denominação quando um sistema cliente-servidor é desenvolvido retirando-se a camada de negócio do lado do cliente. O desenvolvimento é mais demorado no início comparando-se ao modelo em duas camadas porque é necessário dar suporte a uma maior quantidade de plataformas e ambientes diferentes. Em contrapartida, o retorno vem em forma de respostas mais rápidas nas requisições, tanto em sistemas que rodam na Internet ou em intranet, e mais controle no crescimento do sistema.

Índice

1 Definição

2 Camadas

- 2.1 Camada de apresentação
- 2.2 Camada de negócio
- 2.3 Camada de Dados

3 Aplicações

- 3.1 Aplicações monolíticas (uma camada)
- 3.2 Aplicações em duas camadas
- 3.3 Aplicações em três camadas

4 Conclusão

5 Ver também

Definição

As três partes de um ambiente modelo três camadas são: camada de apresentação, camada de negócio e camada de dados. Características esperadas em uma arquitetura cliente-servidor 3 camadas:

- O software executado em cada camada pode ser substituído sem prejuízo para o sistema;
- Atualizações e correções de defeitos podem ser feitas sem prejudicar as demais camadas. Por exemplo: alterações de interface podem ser realizadas sem o comprometimento das informações contidas no banco de dados.

Camadas

Camada de apresentação

É a chamada GUI (Graphical User Interface), ou simplesmente interface. Esta camada interage diretamente com o usuário, é através dela que são feitas as requisições como consultas, por exemplo.

Camada de negócio

Também chamada de lógica empresarial, regras de negócio ou funcionalidade. É nela que ficam as funções e regras de todo o negócio. Não existe uma interface para o usuário e seus dados são voláteis, ou seja, para que algum dado seja mantido deve ser utilizada a camada de dados.

Camada de Dados

É composta pelo repositório das informações e as classes que as manipulam. Esta camada recebe as requisições da camada de negócios e seus métodos executam essas requisições em um banco de dados. Uma alteração no banco de dados alteraria apenas as classes da camada de dados, mas o restante da arquitetura não seria afetado por essa alteração.

Aplicações

Aplicações monolíticas (uma camada)

Nos tempos antigos do reinado do grande porte e do computador pessoal independente, um aplicativo era desenvolvido para ser usado em uma única máquina (standalone). Geralmente esse aplicativo continha todas as funcionalidades em um único módulo gerado por uma grande quantidade de linhas de código e de difícil manutenção. A entrada do usuário, verificação, lógica de negócio e acesso a banco de dados estavam todos presentes em um mesmo lugar.

Aplicações em duas camadas

A necessidade de compartilhar a lógica de acesso a dados entre vários usuários simultâneos fez surgir as aplicações em duas camadas. Nesta estrutura, a base de dados foi colocada em uma máquina específica, separada das máquinas que executavam as aplicações. Nessa abordagem, temos aplicativos instalados em estações clientes contendo toda a lógica da aplicação (clientes ricos ou gordos). Um grande problema neste modelo é o gerenciamento de versões pois, para cada alteração, os aplicativos precisam ser atualizados em todas as máquinas clientes.

Aplicações em três camadas

Com o advento da Internet, houve um movimento para separar a lógica de negócio da interface com o usuário. A ideia é que os usuários da WEB possam acessar as mesmas aplicações sem ter que instalar estas aplicações em suas máquinas locais. Como a lógica do aplicativo, inicialmente contida no cliente rico, não mais reside na máquina do usuário, este tipo de cliente passou a ser chamado de cliente pobre ou magro (Thin Client). Neste modelo o aplicativo é movido para o servidor e um navegador web é usado como um cliente magro. O aplicativo é executado em servidores web com os quais o navegador web se comunica e gera o código HTML para ser exibido no cliente.

Conclusão

No modelo 3 camadas, a lógica de apresentação esta separada em sua própria camada lógica e física. A separação em camadas lógicas torna os sistemas mais flexíveis, permitindo que as partes possam ser alteradas de forma independente. As funcionalidades da camada de negócio podem ser divididas em classes e essas classes podem ser agrupadas em pacotes ou componentes, reduzindo as dependências entre as classes e pacotes; podem ser reutilizadas por diferentes partes do aplicativo e até por aplicativos diferentes. O modelo de 3 camadas tornou-se a arquitetura padrão para sistemas corporativos com base na Web.

Fonte: https://pt.wikipedia.org/wiki/Modelo_em_tr%C3%AAs_camas

Programação de Aplicações simples com Acesso a Bancos de Dados Relacionais

Este material é baseado no conjunto de artigos do site Macoratti.net referentes ao gerenciamento de bancos de dados usando C#. Foi adaptado para incluir o uso do servidor de bancos de dados Sql Server, ao invés do MySql, como constava do artigo original, que pode ser acessado em http://www.macoratti.net/09/08/c_mysql1.htm

C# - Gerenciamento de banco de dados SQL Server

Neste artigo vamos gerenciar um banco de dados SQL Server efetuando as operações *de acesso, seleção, inclusão, alteração e exclusão* usando a linguagem C# e ADO .NET.

ADO.Net é o conjunto de classes do .Net Framework que possibilita o acesso a dados armazenados em bancos de dados relacionais e em outros meios de armazenamento. Vamos trabalhar usando a arquitetura em 3 camadas definindo assim:

- A camada de interface : UI - *namespace UI*
- A camada de negócios : BLL - *namespace BLL e classe produtoBLL*
- A camada de acesso a dados : DAL - *namespace DAL e classe produtoDAL*

Uma arquitetura em 3 camadas significa que haverá 3 classes diferentes para gerenciar as atividades de nosso aplicativo e que cada classe (ou camada) será responsável por um aspecto específico do projeto. Poderia também haver 3 ou mais processadores (diferentes computadores) envolvidos, mas, em nosso caso, teremos apenas dois: o processador do computador que você está usando e o processador do computador em que o servidor de banco de dados está instalado e sendo executado.

Outro conceito importante é o padrão **MVC**, que permite construir aplicações que, mesmo executadas em uma única máquina (ou em duas, como faremos), utiliza conceitos de separação do software em camadas, onde cada camada se responsabiliza por um aspecto do funcionamento do aplicativo.

MVC – Model, View, Controller

MVC não foi criado para ser somente um padrão de projeto, ele na verdade é uma arquitetura de projeto onde seu objetivo é separar seu código em três camadas fazendo com que cada área só trabalhe com itens que competem à elas. Trocando em miúdos, cada um só faz o que foi desenvolvido para fazer. Com o MVC você facilmente transforma seu código de modo à ficar muito mais legível. Para utilizá-lo você tem que ter em mente que haverá uma separação em seu código, as regras de negócio ficarão separadas da lógica e da interface do usuário.

M de Model

O model, ou modelo, no padrão MVC serve para armazenar e persistir os dados. O que seria isso? Toda comunicação com o banco de dados. Os comandos crud (inserir, alterar, remover, buscar) serão feitas pelas classes deste tipo. É utilizado para armazenar informações, trabalhando como um espelho da tabela do banco de dados. Como trabalhamos com objetos, os dados serão persistidos como objetos.

V de View

O view, ou visão, no padrão MVC servirá APENAS para exibir as informações enviadas pelo controller, aqui não existirá nenhuma lógica ou regra de negócio, apenas a interface do usuário.

C de Controller

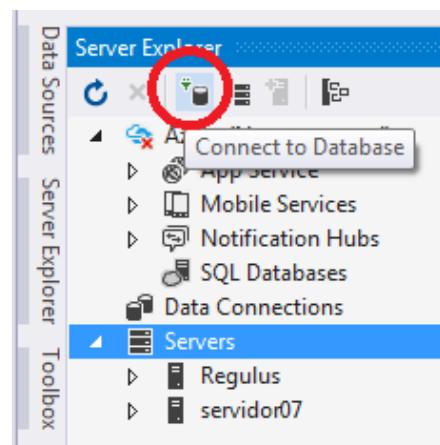
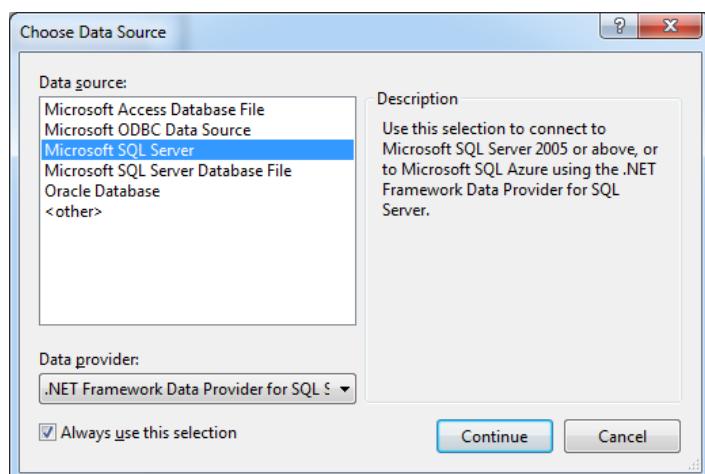
O controle faz exatamente o que o nome diz: controla. Ele é o responsável por fazer o intermédio entre o modelo e a visão. É o responsável também por toda lógica do sistema. Retornando somente os itens necessários para a comunicação entre o modelo e a visão. Entre o usuário e a aplicação.

Nosso aplicativo fará o controle de uma biblioteca, com leitores, títulos e seus empréstimos. Assim, teremos um banco de dados com tabelas Leitor, Titulo e Emprestimo, onde os dados dessas entidades serão armazenados e acessados/tratados pelo nosso programa.

Criação das tabelas da biblioteca no Banco de Dados

Vamos então criar nosso aplicativo e usar os recursos de integração do Visual Studio com o Sql Server para também criar o banco de dados. Crie uma aplicação Windows Forms, chamando-a de apBiblioteca. No lado esquerdo do Visual Studio, acesse a aba Server Explorer e clique no botão Connect Database, como vemos na figura ao lado.

Aparecerá uma janela solicitando o tipo de origem de dados. Selecione Microsoft SQL Server, como vemos abaixo e pressione [Continuar]:

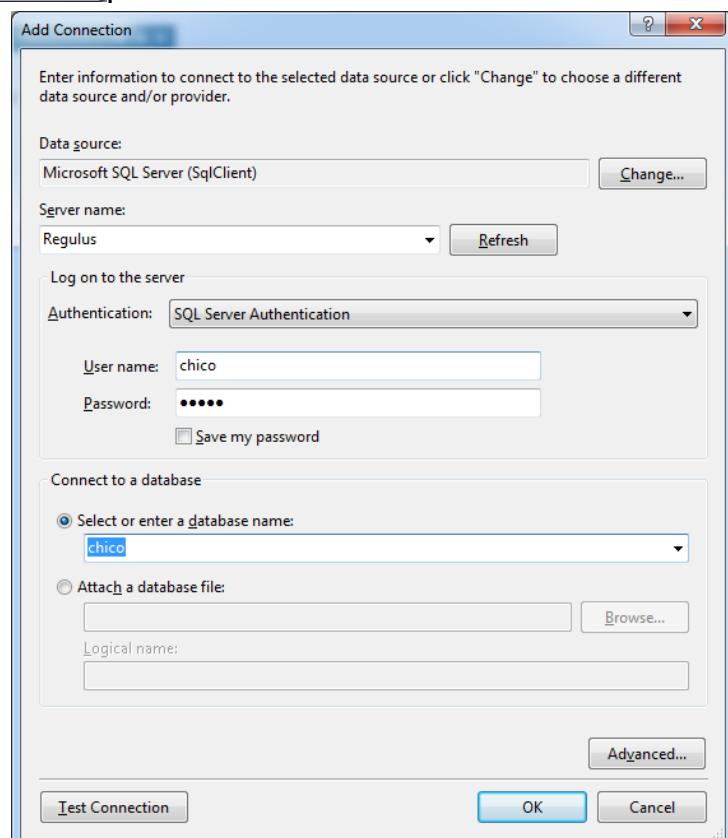


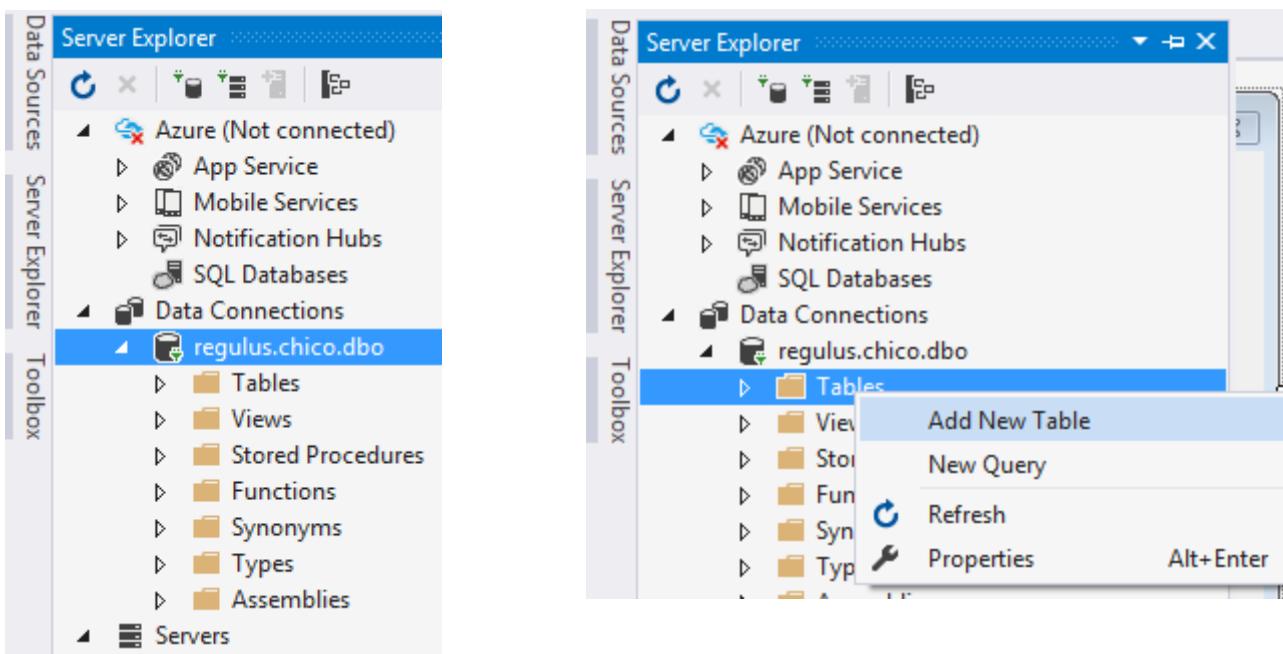
Agora você deverá informar os dados sobre o computador que executa o servidor de banco de dados. No caso de nosso exemplo, o Nome do Servidor é Regulus, utilize SQL Server Authentication e digite seu username e senha para acesso a esse servidor. Selecione também o banco de dados associado à sua conta (em geral, BD<seuRA> (exemplo BD22129)).

Pressione o botão [OK].

Na figura a seguir, veremos que a representação do seu banco de dados apareceu como uma conexão ao servidor de banco de dados Regulus. Nessa estrutura, você poderá acessar as tabelas e demais recursos de seu banco de dados diretamente no Visual Studio, sem necessidade de usar o SQL Server Management Studio.

Clique com o botão direito do mouse sobre a pasta Tables e selecione a opção [Add New Table]:





Monte a descrição da tabela Livro como vemos abaixo. Não se esqueça de selecionar o campo IdLivro e configurar a propriedade **Identity Specification** como vemos na figura, pois esse campo será uma chave primária auto-incremento (Identity).

Name	Data Type	Allow Nulls	Default
IdLivro	int	<input type="checkbox"/>	
codigoLivro	varchar(10)	<input type="checkbox"/>	
tituloLivro	varchar(50)	<input type="checkbox"/>	
autorLivro	varchar(50)	<input type="checkbox"/>	

Properties

IdLivro Column

General

- (Name) **IdLivro**
- Allow Nulls **False**
- Data Type **int**
- Default Value or Bindin
- Description

Table Designer

- Collation
- Computed Column Spec
- Full Text Specification **False**
- Identity Specification **True**
- (Is Identity)** **True**
- Identity Increment **1**
- Identity Seed **1**
- Is Column Set **False**
- Is File Stream **False**
- Is ROWGUID Column **False**
- Is Sparse **False**
- Not For Replication **False**
- Primary Key **True**

(Is Identity)
Specifies whether the column is the identity column for the table.

```

CREATE TABLE [dbo].[Table]
(
    [IdLivro] INT NOT NULL PRIMARY KEY IDENTITY,
    [codigoLivro] VARCHAR(10) NOT NULL,
    [tituloLivro] VARCHAR(50) NOT NULL,
    [autorLivro] VARCHAR(50) NOT NULL
)

```

Na aba T-SQL, mude o nome da tabela para BibLivro, de forma que o comando Create Table fique como abaixo:

```
CREATE TABLE [dbo].[BibLivro]
```

Em seguida, pressione o botão Update sobre a descrição da tabela (parte superior da figura anterior) e, para que essa tabela seja criada, pressione também o botão [Update Database] na janela que será exibida.

Crie agora a tabela de leitores, que se chamará BibLeitor, conforme vemos na figura abaixo:

Name	Data Type	Allow Nulls	Default
IdLeitor	int	<input type="checkbox"/>	
nomeLeitor	varchar(50)	<input type="checkbox"/>	
telefoneLeitor	varchar(20)	<input type="checkbox"/>	
emailLeitor	varchar(50)	<input type="checkbox"/>	
enderecoLeitor	varchar(100)	<input type="checkbox"/>	

IdLeitor Column
Keys (1)
 <unnamed>
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

General
 (Name) IdLeitor
 Allow Nulls False
 Data Type int
 Default Value or Binding
 Description

Table Designer
 Collation

Computed Column Specification (0)

Full Text Specification (0)

Identity Specification True
 (Is Identity) True
 Identity Increment 1
 Identity Seed 1
 Is Column Set False
 Is File Stream False
 Is ROWGUID Column False
 Is Sparse False
 Not For Replication False
 Primary Key True

Em seguida, pressione o botão Update sobre a descrição da tabela (parte superior da figura anterior) e, para que essa tabela seja criada, pressione também o botão [Update Database] na janela que será exibida.

Agora, devemos criar a tabela de Empréstimos, cujo nome será BibEmprestimo.

Sempre que um livro for emprestado a um leitor, deveremos gravar um registro nessa tabela, **relacionando livro com leitor**, armazenando também informações sobre data de empréstimo, data prevista de devolução e data efetiva de devolução.

Cada registro dessa tabela terá um campo Identity que será sua chave primária. Para que possamos relacionar o livro emprestado com o leitor que está levando o livro consigo, temos que guardar no registro a chave primária do livro e a chave primária do leitor. Dessa forma, esses campos serão usados como **chaves estrangeiras** nessa tabela, pois eles se referem à tabela de livros e à tabela de leitores, respectivamente.

A figura abaixo mostra a tela de criação dessa tabela. Após digitar a estrutura da tabela, pressione o botão [Update] e, posteriormente, [Update Database] para que essa tabela seja criada.

The screenshot shows the 'Table Designer' window in SSMS. On the left, there's a grid of columns with headers: Name, Data Type, Allow Nulls, and Default. On the right, the properties for the 'IdEmprestimo' column are displayed, including its type (int), whether it allows nulls (False), its data type (int), and its default value or binding. Below the table grid, the T-SQL code for creating the table is shown:

```

CREATE TABLE [dbo].[BibEmprestimo]
(
    [IdEmprestimo] INT NOT NULL PRIMARY KEY IDENTITY,
    [idLivro] INT NULL,
    [idLeitor] INT NULL,
    [dataEmprestimo] DATETIME NULL,
    [dataDevolucaoPrevista] DATE NULL,
    [dataDevolucaoReal] DATETIME NULL
)

```

Colocamos o prefixo “Bib” no nome das tabelas para que elas fiquem agrupadas na relação de tabelas do banco de dados, como podemos ver na figura ao lado:

Dessa maneira, mesmo que tenhamos vários sistemas diferentes agrupados no mesmo banco de dados, eles serão identificados e agrupados pelo prefixo que escolhermos. Assim, num sistema de bibliotecas, todas as tabelas começam com Bib para que sejam agrupadas e fiquem separadas visualmente das tabelas do sistema financeiro, como prefixo seria, por exemplo, “Fi”.

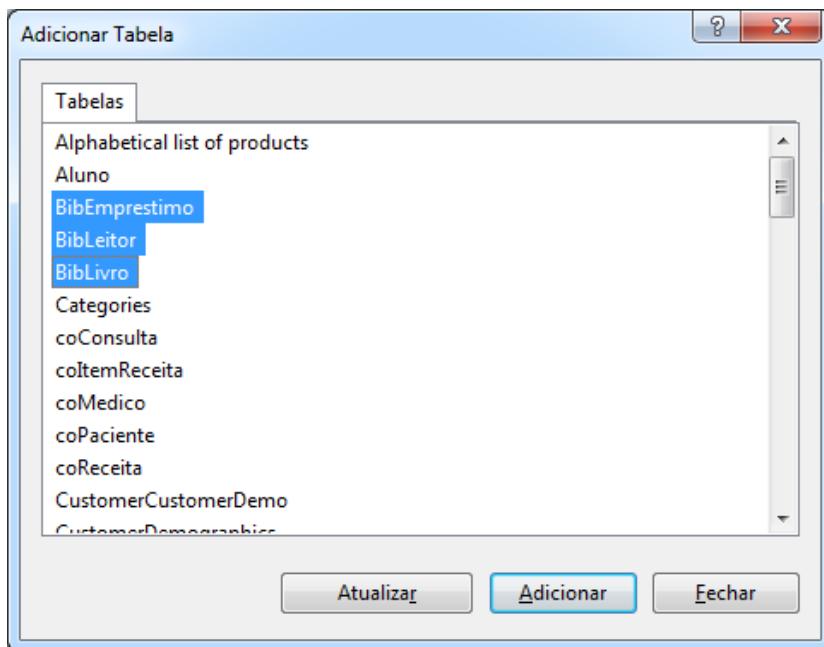
Alternativamente, poderíamos criar um esquema Bib e usar esse esquema para agrupar as tabelas.

Criação dos relacionamentos entre as tabelas

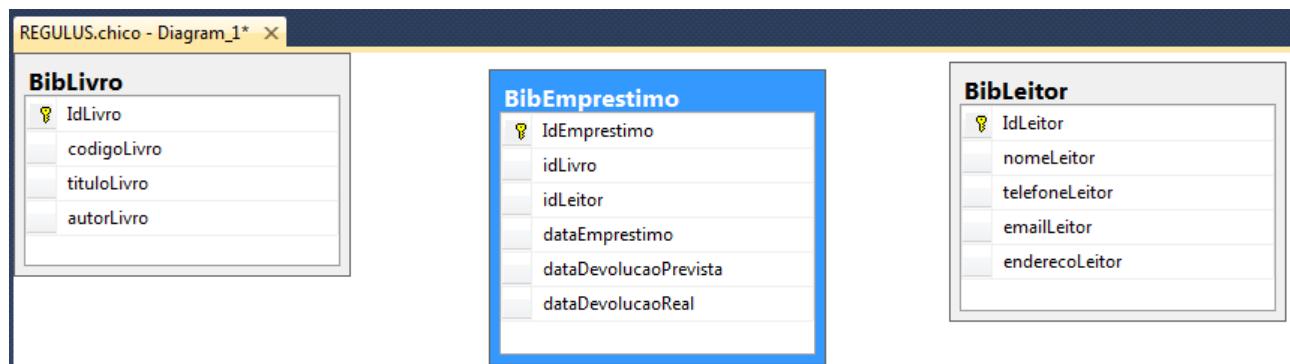
Sabemos que as tabelas somente não bastam para garantir a melhor funcionalidade e a integridade do nosso banco de dados. Precisamos, portanto, criar os relacionamentos que liguem as tabelas e garantam que não haja empréstimos sem livros e sem leitores, por exemplo.

Há uma ferramenta no Sql Server Management Studio que facilita a criação dos relacionamentos entre as tabelas, de forma visual. Abra esse programa, conecte-se ao servidor Regulus e selecione o seu Banco de Dados. Observe que há um item **Diagramas de Banco de Dados**.

Clique com o botão direito nesse item e selecione a opção [Novo Diagrama de Banco de Dados]. Na janela que aparecerá, selecione as três tabelas com prefixo Bib, e pressione o botão [Adicionar], de forma que elas passem a fazer parte do diagrama que criaremos. Em seguida, pressione [Fechar].

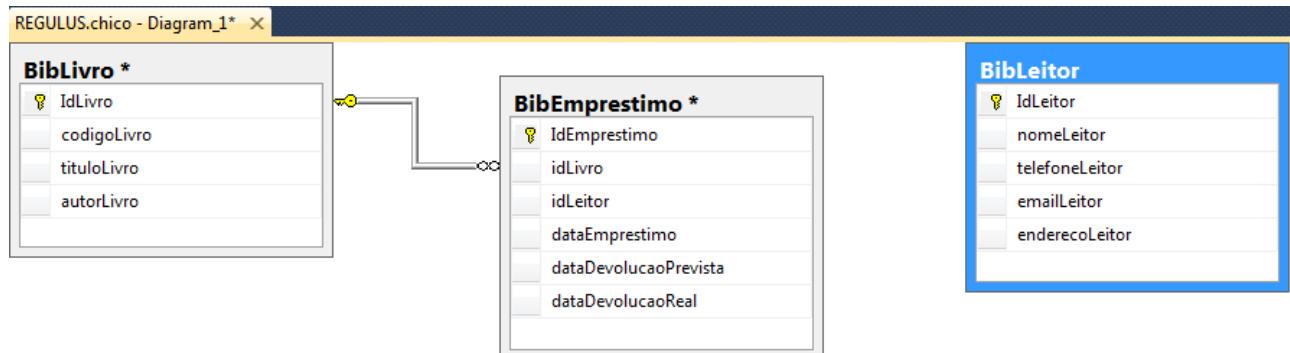


As três tabelas serão exibidas na janela do diagrama. Arraste as tabelas de forma que BibEmprestimo fique entre as outras duas. Isso é feito para facilitar a visualização das ligações entre as tabelas (seu relacionamentos)



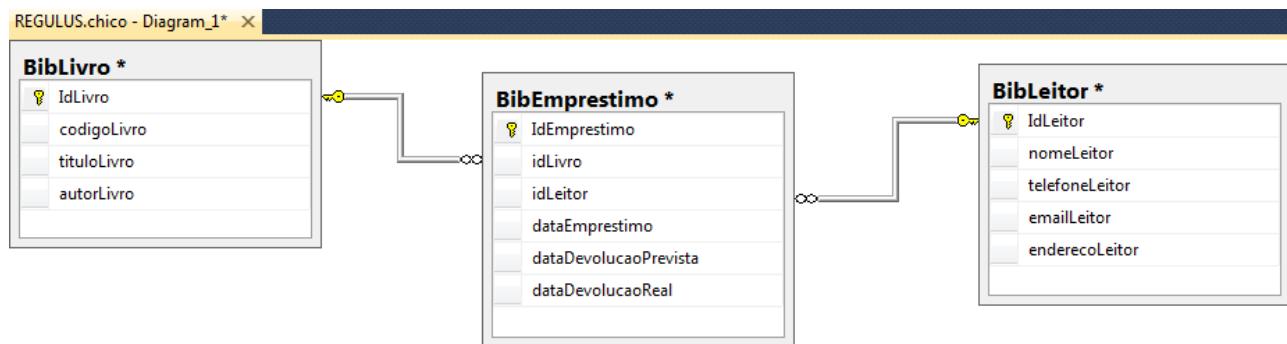
Sabemos que a tabela BibEmprestimo tem dois campos que são chaves estrangeiras (idLivro e idLeitor) que referenciarão as outras duas tabelas, através da ligação entre os campos.

Portanto, clique no quadrinho do lado esquerdo de idLivros da tabela BibEmprestimo e o arraste até o mesmo campo na tabela BibLivro: Pressione [Ok] até que a janela fique como abaixo:

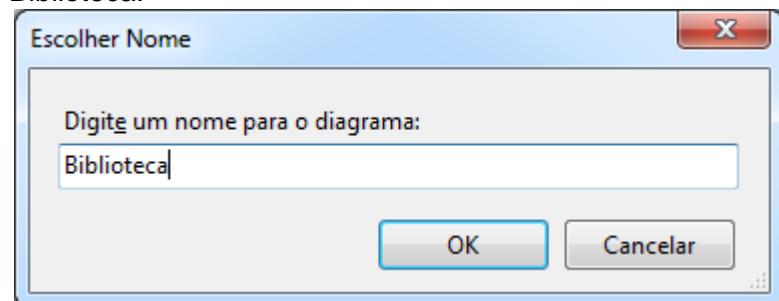


Esse processo criou a Constraint que efetiva o relacionamento entre Emprestimo e Livro, de forma que um empréstimo só pode ocorrer se o idLivro do Emprestimo existir na tabela BibLivro. Você poderia, alternativamente, usar um script no SSMS com o comando Alter Table para incluir as constraints.

Faça agora a ligação entre idLeitor de BibEmprestimo com idLeitor de BibLeitor. O diagrama ficará como abaixo:



Clique no botão [Salvar] para que o diagrama seja salvo. Será solicitado o nome do diagrama. Digite Biblioteca.



Após realizar esta tarefa, publique no Google Classroom que ela foi finalizada, na tarefa que será criada para tanto. Entregue no prazo. O seu banco de dados será avaliado para verificação da criação das tabelas e seus relacionamentos.

Criando as classes para representar as entidades do Banco de Dados

Criaremos inicialmente a classe Livro. Isso também será feito no Visual Studio, no aplicativo apBiblioteca.

Para isso, no Gerenciador de Soluções clique com o botão direito no nome do projeto, selecione a opção [Adicionar] e, em seguida, [Classe].

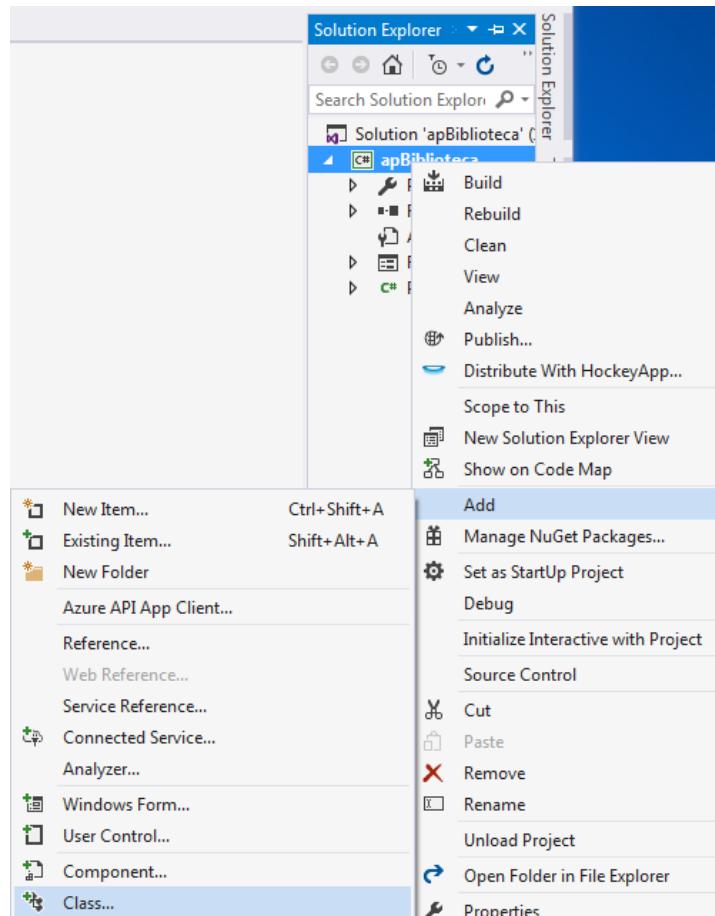
Digite Livro como o nome da classe e declare os atributos da classe, que devem refletir os campos da tabela bibLivre, ou seja, idLivro, codigoLivro, tituloLivro e autorLivro:

```

class Livro
{
    const int tamanhoCodigo = 6;
    const int tamanhoTitulo = 50;
    const int tamanhoAutor = 50;

    int idLivro;
    string codigoLivro;
    string tituloLivro;
    string autorLivro;
}
  
```

Note a declaração de constantes com os tamanhos máximos de cada campo,



conforme foram definidos na tabela bibLivro do Banco de Dados.

É importante que a classe Livro, que representará a tabela de livros em nosso aplicativo, seja consistente com as declarações do Banco de Dados, para que não haja erros de gravação. Por exemplo, se o usuário pudesse digitar mais de 6 caracteres no código do livro, no momento da gravação de um registro de livro na tabela de livro (insert into BibLivro...) haveria erro e o servidor de Banco de Dados abortaria o comando de inserção, gerando uma exceção que poderia abortar o funcionamento de nosso aplicativo.

Portanto, as propriedades abaixo farão com que os tamanhos máximos sejam respeitados:

```

public int IdLivro
{
    get => idLivro;
    set
    {
        if (value < 0)
            throw new Exception("Id negativo é inválido!");
        idLivro = value;      // armazena o valor passado no atributo de destino
    }
}

public string CódigoLivro
{
    get => códigoLivro;
    set
    {
        // remove qualquer caractere além do tamanho máximo do campo
        value = value.Remove(tamanhoCódigo);
        // preenche código com zeros à esquerda até completar o tamanho máximo
        value = value.PadLeft(tamanhoCódigo, '0');
        // armazena o valor passado no atributo de destino
        códigoLivro = value;
    }
}

public string TítuloLivro
{
    get => títuloLivro;
    set
    {
        // remove qualquer caractere além do tamanho máximo do campo
        value = value.Remove(tamanhoTítulo);
        // preenche título com espaços à direita até completar o tamanho máximo
        value = value.PadRight(tamanhoTítulo, ' ');
        // armazena o valor passado no atributo de destino
        títuloLivro = value;
    }
}

public string AutorLivro
{
    get => autorLivro;
    set
}

```

```

{
    // remove qualquer caracter além do tamanho máximo do campo
    value = value.Remove(tamanhoAutor);

    // preenche título com espaços à direita até completar o tamanho máximo
    value = value.PadRight(tamanhoAutor, ' ');

    // armazena o valor passado no atributo de destino
    autorLivro = value;
}

}

```

As propriedades serão usadas para impedir que a aplicação tenha acesso direto aos atributos da classe. Elas permitem disciplinar o acesso aos atributos, de modo que consistências (verificações) sejam feitas e que se controle esse acesso, para evitar problemas no banco de dados.

Temos agora que criar o construtor da classe Livro, como vemos abaixo:

```

public Livro(int id, string codigo, string titulo, string autor)
{
    IdLivro = id;
   CodigoLivro = codigo;
    TituloLivro = titulo;
    AutorLivro = autor;
}

```

Observe que, no construtor acima, usamos os nomes das propriedades (inicial em maiúscula) e não os nomes dos atributos (inicial em minúscula). Isso fará com que, quando uma propriedade (por exemplo, CódigoLivro) receber um valor no comando de atribuição, seja invocado o acessador set dessa propriedade. Assim, os comandos que foram codificados na parte set da propriedade serão executados e os tamanhos máximos dos campos da tabela serão respeitados.

Exercício

1. Usando a discussão e a classe anteriores como modelo, crie as classes referentes às tabelas Leitor e Emprestimo, que também serão usadas em nosso aplicativo.

Criação das classes da Camada de Acesso a Dados

Estamos usando a arquitetura em 3 camadas definida assim:

- A camada de acesso a dados : DAL - namespace *DAL* e classe *produtoDAL*
- A camada de negócios : BLL - namespace *BLL* e classe *produtoBLL*
- A camada de interface : UI - namespace *UI*

Em uma aplicação em 3 camadas temos uma hierarquia de chamadas onde:



As camadas acima são também conhecidas por:

- Camada de apresentação – UI (user interface)
- Camada de lógica de negócios – BLL (Business Logic Layer)
- Camada de Persistência ou de Acesso a Dados – DAL (Data Access Layer)

A UI chama a BLL que chama a DAL que, por sua vez, **acessa os dados e retorna os objetos**.

Nunca deverá haver uma chamada direta da UI para a DAL e vice-versa.

Quando usamos os controles de acesso a dados vinculados no formulário estamos fazendo o acesso direto da camada de interface para a DAL ou banco de dados, o que não é uma boa prática.

Nosso projeto possui a seguinte estrutura:

- Livro - contém a classe Produto; (*namespace DTO*)
- LivroDAL - Contém os métodos para acesso a dados no SQL Server; (*namespace DAL*)
- LivroBLL - contém os métodos das regras de negócio; (*namespace BLL*)
- UI - representa a nossa aplicação Windows Forms;

Crie as seguintes pastas no Gerenciador de Soluções: BLL, DAL, DTO e UI.

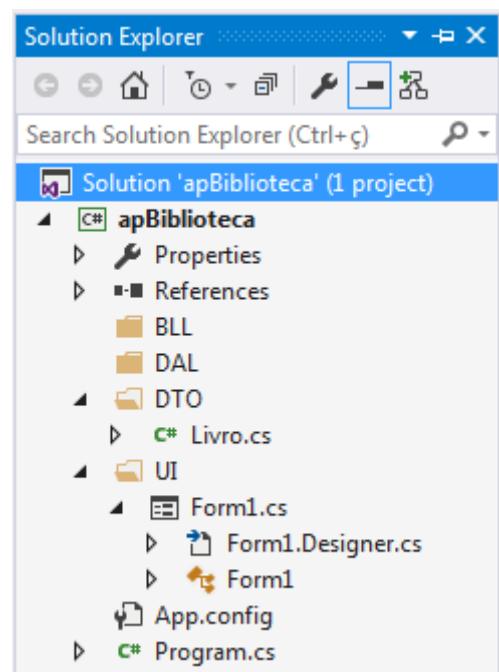
DTO é uma sigla para Data Transfer Object (Objeto de Transferência de Dados). DTO é um padrão de projeto de software usado para transferir dados entre subsistemas de um software. **DTOS** são frequentemente usados em conjunção com objetos da camada de acesso a dados para obter e armazenar dados de um banco de dados.

Arraste Form1.cs para UI e Livro.cs para DTO. O gerenciador de soluções ficará como na figura ao lado:

Após ter feito o exercício da seção anterior, arraste os arquivos Leitor.cs e Emprestimo.cs para a pasta DTO.

Agora vamos criar o arquivo de classe LivroDAL.cs clicando no nome da pasta DAL e selecionando a opção Add New Item;

Na janela de *templates* selecione o *template Class* e informe o nome LivroDAL.cs;



Os namespaces (pacotes e classes prontos) usados nesta classe são:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
```

A seguir vamos definir na classe LivroDAL sete métodos conforme exibidos abaixo:

1. **SelectListLivros** - retorna uma lista de Livros, ou seja, um objeto da classe List<Livro> com os livros selecionados;
2. **SelectLivros** - retorna um DataTable com os livros selecionados;
3. **SelectLivroByID** - retorna uma entidade Livro para um livro selecionado pelo seu **idLivro**;
4. **SelectLivroByCodigo**- retorna uma entidade Livro para um livro selecionado pelo seu **codigoLivro**;
5. **InsertLivro** - inclui um novo livro;
6. **UpdateLivro** - atualiza um livro existente;
7. **DeleteLivro** - exclui um livro existente

Abaixo temos o código inicial dessa nova classe:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using DTO;

namespace DAL
{
    public class LivroDAL
    {
        string _conexaoSQLServer = "";
        SqlConnection _conexao = null;

        public LivroDAL(string banco, string usuario, string senha)
        {
            _conexaoSQLServer =
                $"Data Source=regulus.cotuca.unicamp.br; Initial Catalog={banco};" +
                $"User id={usuario}; Password={senha}";
            _conexao = new SqlConnection(_conexaoSQLServer);
        }

        public List<Livro> SelectListLivros()...
```

Vamos agora codificar os métodos internamente:

SelectLivros() – usado para retornar uma tabela de dados com os registros da tabela Livro

```
public DataTable SelectLivros()
{
    try
    {
        string sql="SELECT idLivro,codigoLivro,tituloLivro,autorLivro FROM bibLivro";
        SqlCommand cmd = new SqlCommand(sql, _conexao);
        _conexao.Open();
        SqlDataAdapter da = new SqlDataAdapter();
        da.SelectCommand = cmd;
        DataTable dt = new DataTable();
        da.Fill(dt);
        _conexao.Close();
        return dt;
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

SelectLivroById() - Usada para retornar uma entidade Livro representando um único livro pela sua chave primária (idLivro):

```
public Livro SelectLivroById(int id)
{
    try {
        string sql = "SELECT idLivro, codigoLivro, tituloLivro, autorLivro "+
                    " FROM bibLivro WHERE idLivro = @id";
        SqlCommand cmd = new SqlCommand(sql, _conexao);
        cmd.Parameters.AddWithValue("@id", id);
        _conexao.Open();
        SqlDataReader dr;
        dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
        Livro livro = null;
        if (dr.Read())
        {
            livro = new Livro(Convert.ToInt32(dr["idLivro"]),
                               dr["codigoLivro"].ToString(),
                               dr["tituloLivro"].ToString(),
                               dr["autorLivro"].ToString());
        }
        _conexao.Close();
        return livro;
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

SelectLivroByCodigo - retorna uma entidade Livro para um livro definido pelo seu codigoLivro

```
public Livro SelectLivroByCodigo(string codigo)
{
    try {
        string sql = " SELECT idLivro, codigoLivro, tituloLivro, autorLivro " +
                     " FROM bibLivro WHERE codigoLivro = @codigo";
        var cmd = new SqlCommand(sql, _conexao);
        cmd.Parameters.AddWithValue("@codigo", codigo);
        _conexao.Open();
        SqlDataReader dr;
        dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
        Livro livro = null;
        if (dr.Read())
            livro = new Livro(Convert.ToInt32(dr["idLivro"]),
                               dr["codigoLivro"].ToString(),
                               dr["tituloLivro"].ToString(),
                               dr["autorLivro"].ToString());
        _conexao.Close();
        return livro;
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

SelectListLivros() - Usada para retornar uma lista de objetos Livro representando uma coleção de livros.

```
// retorna uma lista de Livros, ou seja, um objeto da classe List<Livro> com os
// livros selecionados;
public List<Livro> SelectListLivros ()
{
    try {
        var cmd = new SqlCommand("Select * from bibLivro", _conexao);
        _conexao.Open();
        var listaLivros = new List<Livro>();
        var dr = cmd.ExecuteReader();
        while (dr.Read())
        {
            var livro = new Livro( (int)dr["idLivro"],
                                  dr["codigoLivro"] + "", 
                                  dr["tituloLivro"] + "", 
                                  dr["autorLivro"] + ""
                                );
            listaLivros.Add(livro);
        }
        _conexao.Close();
        return listaLivros;
    }
    catch (Exception ex)
    {
        throw new Exception("Erro ao acessar livro " + ex.Message);
    }
}
```

InsertLivro() – inclui um novo livro na tabela de Livros do Banco de Dados

```
public void InsertLivro(Livro qualLivro)
{
    try {
        string sql = "INSERT INTO bibLivro "+
                    " (codigoLivro, tituloLivro, autorLivro) "+
                    " VALUES (@codigo,@titulo, @autor) ";
        SqlCommand cmd = new SqlCommand(sql, _conexao);
        cmd.Parameters.AddWithValue("@codigo", qualLivro.CodigoLivro);
        cmd.Parameters.AddWithValue("@titulo", qualLivro.TituloLivro);
        cmd.Parameters.AddWithValue("@autor", qualLivro.AutorLivro);
        _conexao.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    finally
    {
        _conexao.Close();
    }
}
```

UpdateLivro() - atualiza um livro existente, cujo idLivro vem no objeto qualLivro. Os novos valores são aqueles armazenados nos demais atributos do objeto qualLivro:

```
public void UpdateLivro(Livro qualLivro)
{
    try {
        string sql = "UPDATE bibLivro "+
                    " SET tituloLivro= @titulo, codigoLivro=@codigo,"+
                    " autorLivro=@autor "+
                    " WHERE idLivro = @idLivro ";
        SqlCommand cmd = new SqlCommand(sql, _conexao);
        cmd.Parameters.AddWithValue("@idLivro", qualLivro.IdLivro);
        cmd.Parameters.AddWithValue("@codigo", qualLivro.CodigoLivro);
        cmd.Parameters.AddWithValue("@titulo", qualLivro.TituloLivro);
        cmd.Parameters.AddWithValue("@autor", qualLivro.AutorLivro);
        _conexao.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    finally
    {
        _conexao.Close();
    }
}
```

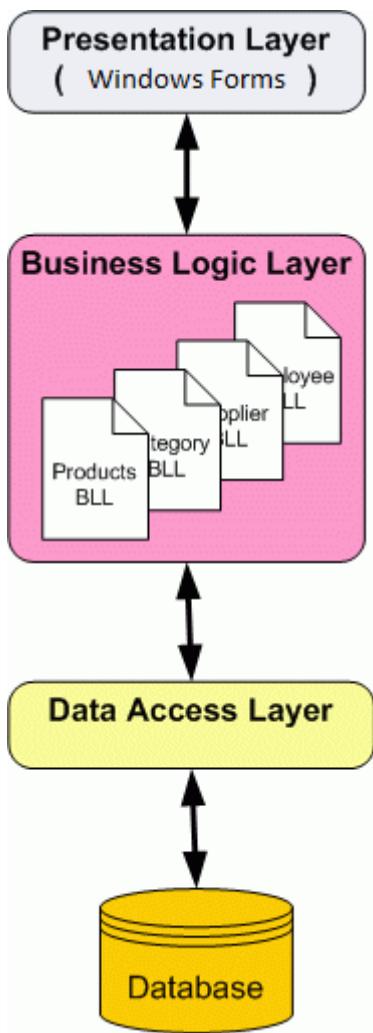
DeleteLivro() – exclui um livro existente, cujo idLivro vem no objeto qualLivro:

```
public void DeleteLivro(Livro qualLivro)
```

```
{  
    try  
    {  
        String sql = "DELETE FROM bibLivro WHERE idLivro = @idLivro ";  
        SqlCommand cmd = new SqlCommand(sql, _conexao);  
        cmd.Parameters.AddWithValue("@idLivro", qualLivro.IdLivro);  
        _conexao.Open();  
        cmd.ExecuteNonQuery();  
    }  
    catch (Exception ex)  
    {  
        throw ex;  
    }  
    finally  
    {  
        _conexao.Close();  
    }  
}
```

Codifique essa classe e a use como modelo para criar as classes LeitorDAL e EmprestimoDAL. Esta é a nossa camada de acesso a dados e sua responsabilidade é acessar e persistir dados no SQL Server.

Criação das classes da Camada de Lógica de Negócio



A camada Data Access Layer (DAL) claramente **separa** a lógica de acesso aos dados da lógica de apresentação (formulário Windows, Formulário Web ou dispositivo móvel, por exemplo). No entanto, enquanto a DAL não força nenhuma regra de negócio que possam ser necessárias, pois se preocupa apenas com o acesso e busca dos dados. Por exemplo, para nossa aplicação podemos querer desabilitar o empréstimo de um livro quando este já estiver emprestado a um leitor ou impedir que um leitor faça novos empréstimos se possuir consigo um certo número de livros ou tiver livros em atraso. Outro cenário comum é autorização – talvez somente usuários de um determinado papel possam apagar livros ou renovar datas de devolução de livros já em atraso.

Nossa Camada de Lógica de Negócio (BLL) será criada na pasta BLL da nossa aplicação e conterá uma classe para cada uma das tabelas TableAdapter da camada DAL. Cada uma dessas classes da BLL terão métodos para recuperar, inserir, atualizar e remover registros da respectiva TableAdapter da DAL, aplicando as regras de negócio apropriadas para que o funcionamento correto da Biblioteca seja garantido.

Como fizemos anteriormente, iniciaremos a criação das classes pela entidade Livro. Crie uma nova classe, chamada LivroBLL e nela codifique os métodos abaixo:

- DataTable SelecionarLivros() - retorna todos os livros;
- IncluirLivro(Livro livro) - inclui um novo livro;
- AlterarLivro(Livro livro) - altera os dados de um livro;
- ExcluirLivro(Livro livro) - exclui um livro;
- List<Livro> ListarLivros() - retorna uma lista de livros;
- ListarLivroPorID(int id) - retorna um único livro;
- ListarLivroPorCodigo(string código) – idem.

Esses métodos serão chamados pela camada de apresentação (o formulário Windows) e, por sua vez, chamarão métodos descritos na DAL, que fará o acesso aos recursos do Banco de Dados.

Abaixo temos as assinaturas desses métodos, já implementados em C#. Observe os comandos using que utilizam Collection e Data.

```

using System;
using System.Collections.Generic;
using System.Data;
using DAL;
using DTO;

namespace BLL
{
    class LivroBLL
    {
        public string banco, usuario, senha;
        LivroDAL dal = null;
        public LivroBLL(string banco, string usuario, string senha)
        {
            this.banco = banco;
            this.usuario = usuario;
            this.senha = senha;
        }
    }
}

```

```

    }
    public DataTable SelecionarLivros() { }
    public void IncluirLivro(Livro livro) { }
    public void AlterarLivro(Livro livro) { }
    public void ExcluirLivro(Livro livro) { }
    public List<Livro> ListarLivros() { }
    public Livro ListarLivroPorId(int id) { }
    public Livro ListarLivroPorCodigo(string codigo) { }
}
}

```

Agora, vejamos os métodos:

1. **SelecionarLivros()** – retorna um DataTable com todos os livros usando o método **SelectLivros()** da camada DAL:

```

public DataTable SelecionarLivros()
{
    DataTable tb = new DataTable();
    try
    {
        dal = new DAL.LivroDAL(banco, usuario, senha);
        tb = dal.SelectLivros();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    return tb;
}

```

2. **ListarLivroPorId(int id)** - Usada para retornar uma entidade Livro, que representa um único Livro buscado pela chave primária identity (id), através da chamada ao método **SelectLivroByID()** da camada DAL:

```

public Livro ListarLivroPorId(int id)
{
    try
    {
        dal = new DAL.LivroDAL(banco, usuario, senha);
        return dal.SelectLivroByID(id);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}

```

3. **ListarLivroPorCodigo(string codigo)** - Usada para retornar uma entidade Livro, que representa um único Livro buscado pelo seu código, através da chamada ao método **SelectLivroByCodigo()** da camada DAL:

```

public Livro ListarLivroPorCodigo(string codigo)
{
    try
    {
        dal = new DAL.LivroDAL(banco, usuario, senha);
        return dal.SelectLivroByCodigo(codigo);
    }
}

```

```

        }
        catch (Exception ex)
        {
            throw ex;
        }
    }
}

```

4. **ListarLivros()** - Usada para retornar uma lista de objetos Livro representando uma coleção de produtos usando o método **SelectListLivros ()** da camada DAL;

```

public List<Livro> ListarLivros()
{
    try
    {
        dal = new DAL.LivroDAL(banco, usuario, senha);
        return dal.SelectListLivros();
    }
    catch (Exception ex)
    {
        throw ex;
    }
}

```

5. **IncluirLivro(Livro livro)** - Usada para incluir um novo livro no estoque usando o método **InsertLivro()** da camada DAL:

```

public void IncluirLivro(Livro livro)
{
    try
    {
        dal = new DAL.LivroDAL(banco, usuario, senha);
        dal.InsertLivro(livro);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}

```

6. **AlterarLivro()** - Usada para atualizar os dados de um livro no acervo através do método **UpdateLivro()** da camada DAL:

```

public void AlterarLivro(Livro livro)
{
    try
    {
        dal = new DAL.LivroDAL(banco, usuario, senha);
        dal.UpdateLivro(livro);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}

```

7. **ExcluirLivro()** - Usada para excluir um livro do acervo via método **DeleteLivro()** da camada DAL:

```
public void ExcluirLivro(Livro livro)
{
    try
    {
        dal = new DAL.LivroDAL(banco, usuario, senha);
        dal.DeleteLivro(livro);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

Não estamos efetuando nenhuma validação de negócio nessa classe devido a simplicidade do exemplo, mas, em um sistema mais complexo, aqui teríamos as **validações referentes ao negócio como restrições de valores, cálculo de impostos, descontos, etc.**

No caso da tabela de Emprestimo, esse tipo de validação tem que ser feito pois, como dissemos no início desta seção, existem restrições quanto a empréstimos.

Dessa forma concluímos a definição do código da nossa camada de negócios - BLL - através da implementação dos métodos da classe LivroBLL.

Codifique essa classe e a use como modelo para criar as classes LeitorBLL e EmprestimoBLL, das entidades Leitor e Emprestimo, respectivamente. Durante a codificação, fique atento a qualquer **regra de negócio que venha a ser necessária. Essas regras devem ser implantadas na classe de BLL da entidade.**

Codifique comentários explicando as regras de negócio que você detectou e implantou em seu código de BLLs.

Esta é a nossa camada de lógica de negócio e sua responsabilidade é garantir que os dados sejam acessados e consistidos em termos de seus requisitos e restrições.

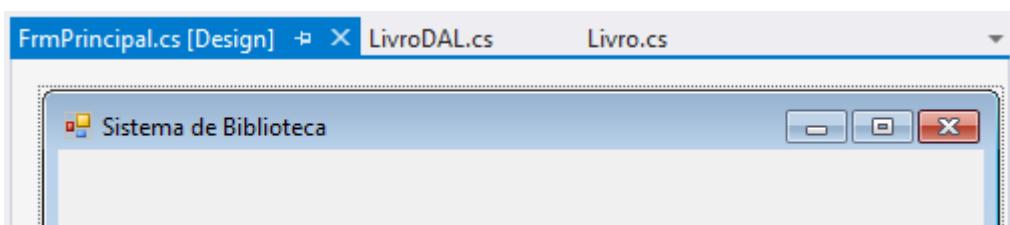
Acessar e apresentar as informações na camada de apresentação

Vamos agora começar a criar a Camada de Apresentação de nosso aplicativo de Biblioteca. Isso será feito através de uma aplicação Windows Forms, onde utilizaremos conceitos de programação visual, acesso a bancos de dados e orientação a objetos, utilizando as classes que criamos anteriormente (DTO, DAL e BLL) bem como os formulários Windows do C#, que também são classes.

Primeiramente, devemos criar um formulário principal, com um menu que permitirá chamar os demais formulários (Livros, Leitores e Operações).

No Visual Studio, vamos abrir nossa aplicação apBiblioteca, que criamos logo no início deste estudo, como uma aplicação Windows Forms.

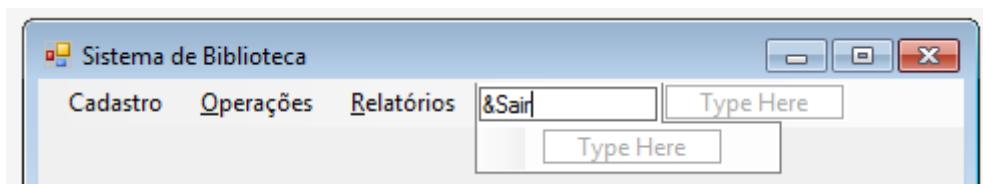
O único formulário da nossa aplicação ainda está vazio. No Gerenciador de Soluções, apague o arquivo Form1.cs e crie um novo formulário, chamando-o de FrmPrincipal.cs. Mude a propriedade Text desse formulário para “Sistema de Biblioteca”:



Vamos agora criar o menu de nosso aplicativo, seguindo a explicação a seguir sobre menus.

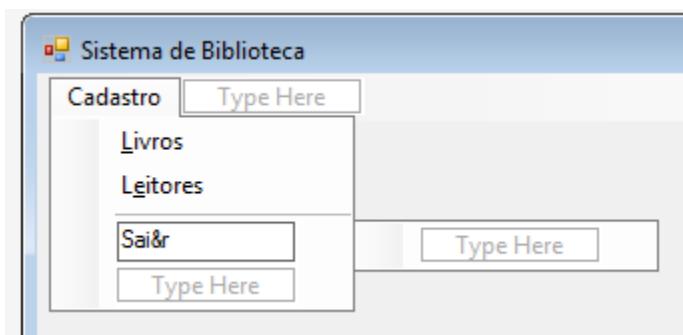
Um menu é um componente extremamente fácil de se projetar e muito útil. Permite prover ao usuário maneiras alternativas de executar operações. O Visual Studio fornece um projetista de menus, que é ativado a partir da colocação, no formulário, de um componente **MenuStrip**, encontrado na coleção **Menus e Barras de Ferramentas** da Caixa de Ferramentas.

Após ser colocado, pode-se indicar as opções do primeiro nível do menu.



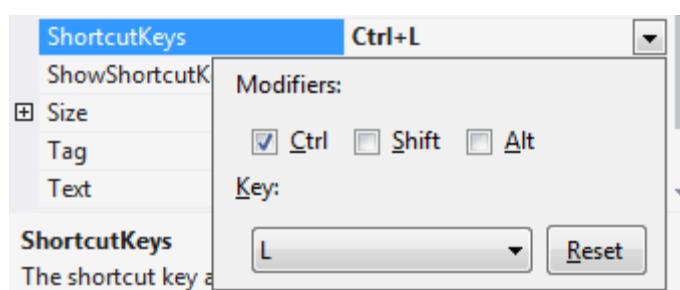
Após digitar o texto de uma opção do Menu, pode-se pressionar a tecla [Tab] e [Enter] para mudar para a opção da direita e abrir a caixa de digitação do texto da mesma. Na figura acima, depois de digitar o texto “&Relatórios”, pressionar [Tab] mudará o cursor para a opção à direita e pressionar [Enter] abrirá a caixa de digitação do texto dessa opção. Também pode-se clicar com o cursor do mouse na caixa de digitação do texto, diretamente.

Se você clicar na caixa de digitação abaixo da opção atual, abrirá um sub-menu, que será apresentado na tela quando a opção for selecionada e você poderá digitar mais opções subordinadas à opção superior, como vemos na figura ao lado.

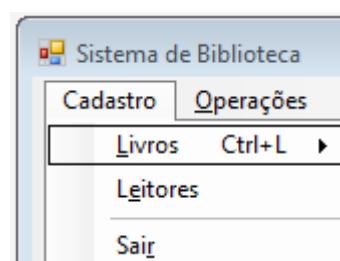


Para criar a linha horizontal separando áreas do menu vertical, você deve digitar o caracter hífen (“-”) no texto da opção, fazendo aparecer essa linha.

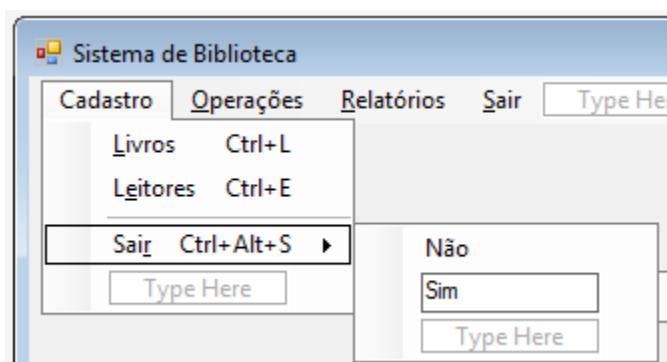
Você também pode associar uma tecla de atalho a um item de menu. Por exemplo, selecione o item Livros e, na janela de propriedades, acesse a propriedade ShortcutKeys e marque Ctrl e L:



Ficará assim:



Você pode criar sub-menus laterais para um item do menu vertical, como vemos na figura abaixo:



Para associar código executável a uma opção de menu, deve-se codificar o evento **Click** de cada uma. Por exemplo, se houvesse uma opção Sobre do menu principal acima, pode-se escrever o código `FrmSobre.Show()`, supondo que há um outro formulário com esse nome, no evento Click desse item de menu:

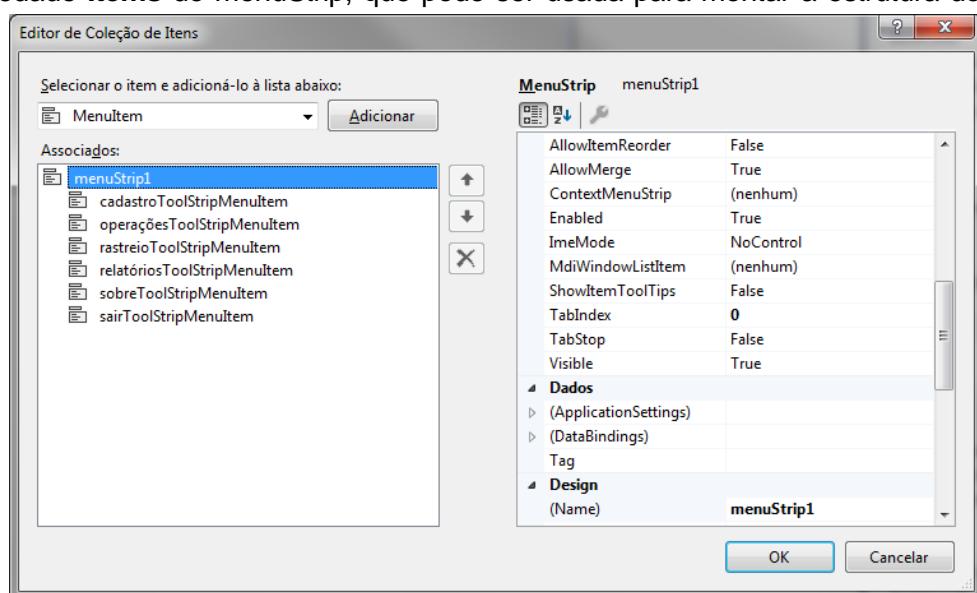
```
frmSobre.Show();
```

Isso fará com que o formulário seja exibido e poderá ter suas funcionalidades executadas.

No evento Click do item Sair do menu, podemos executar o método Close() do formulário atual, de forma que o programa tenha sua execução finalizada.

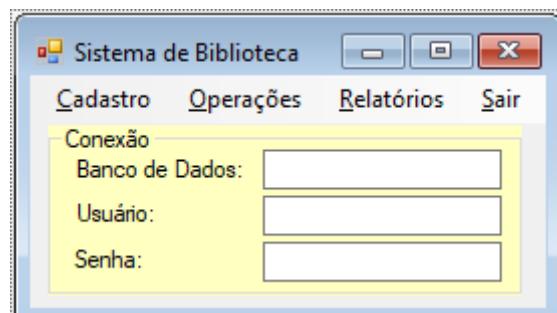
Ou seja, cada item do menu pode ter um evento Click associado e, no método tratador desse evento, qualquer comando pode ser executado, inclusive com a exibição de outros formulários.

Existe também a propriedade **Items** do menuStrip, que pode ser usada para montar a estrutura do menu, como vemos na figura abaixo. Explore-a e verifique como ela funciona, pois pode ser uma ferramenta interessante para auxiliar a criação do menu do seu aplicativo. Abaixo temos um exemplo com várias opções de **um outro projeto**:



No interior do formulário, inclua os controles Groupbox, Label e Textbox da figura ao lado. Os textboxes se chamam, respectivamente, txtBanco, txtUsuario e txtSenha. No controle txtSenha mude a propriedade PasswordChar para *.

Os dados digitados nesses controles serão usados para informar à BLL e à DAL os parâmetros para a cadeia de conexão ao banco de dados.



Chamando os demais formulários

No Gerenciador de Soluções, adicione na pasta UI três formulários, chamados FrmLivro, FrmLeitor e FrmOperacoes.

Clique duas vezes no **item de menu** Livros para criar o tratador desse evento. Digite o código descrito dentro das caixas abaixo:

```
using UI;
...
public partial class FrmPrincipal : Form
{
    FrmLivro frmLivro = null;
    FrmLeitor frmLeitor = null;
    FrmOperacoes frmOperacoes = null;
...
    private void livrosToolStripMenuItem_Click(object sender, EventArgs e)
    {
        if (txtBanco.Text == "" || txtUsuario.Text == "" ||
            txtSenha.Text == "")
            MessageBox.Show("Preencha os dados de conexão!");
    }
}
```

```

else
{
    frmLivro = new UI.FrmLivro();
    frmLivro.banco = txtBanco.Text;
    frmLivro.usuario = txtUsuario.Text;
    frmLivro.senha = txtSenha.Text;
    frmLivro.Show();
}
}

```

Vamos agora dar continuidade ao projeto do formulário de Livros. Abra esse formulário na janela de design e crie uma interface como a que vemos abaixo, onde usamos um **TabControl** com duas **TabPage**s. Após colocar uma TabControl no formulário, você criar TabPages clicando com o botão direito na TabControl e selecionando a opção [Add Tab].

Na primeira TabPage teremos a visão com detalhes do livro e, na outra, teremos uma relação de todos os livros cadastrados num componente DataGridView.

Ancore o TabControl nos lados Direito e Fundo do formulário, usando a propriedade Anchor.

Mude Text de tabPage1 para Cadastro e de tabPage2 para Lista.

No tabPage1 coloque os itens que vemos na lista e figura abaixo:

FrmLivro – Text: Manutenção de Livros

4 Labels com os Text Identificação, “Código do Livro”, “Título do Livro”, “Autor(es) do Livro”

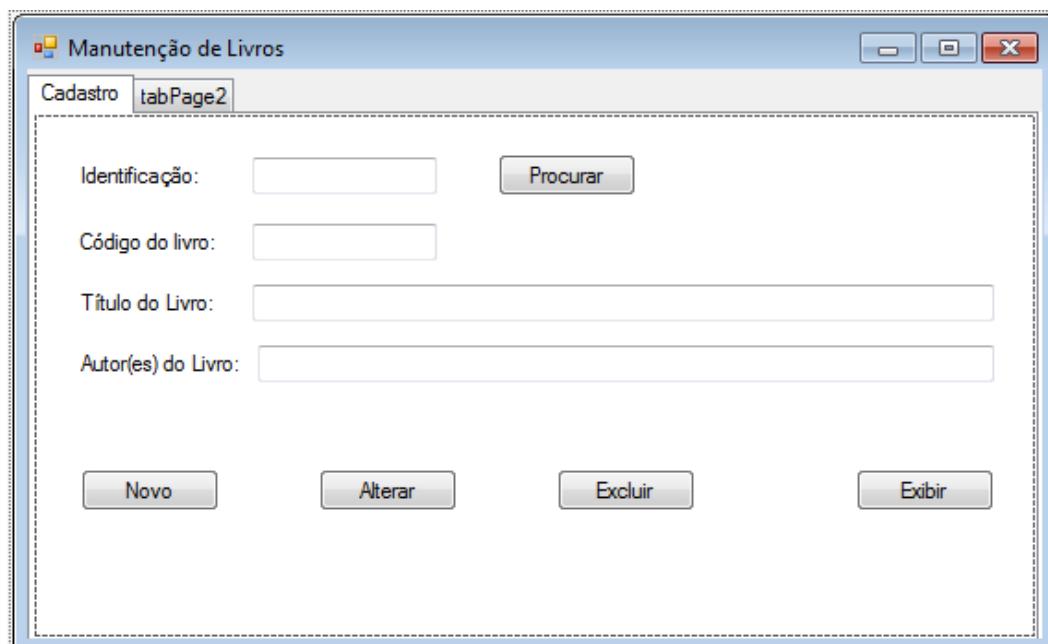
4 TextBox com nomes txtIdLivro, txtCodigoLivro, txtTituloLivro, txtAutorLivro

txtCodigoLivro – MaxLength = 10

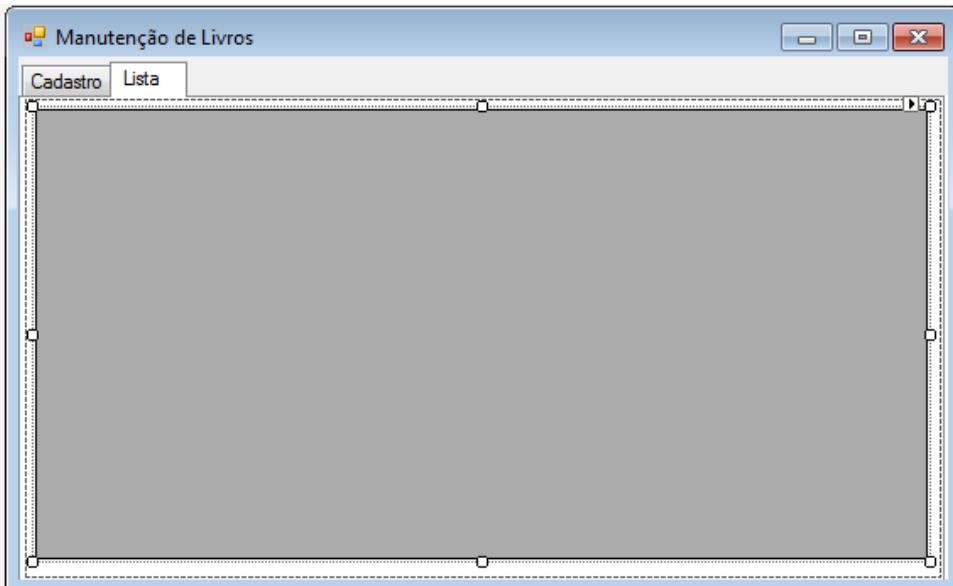
txtTituloLivro – MaxLength = 50

txtAutorLivro – MaxLength = 50

5 botões com nomes btnProcurar, btnNovo, btnAlterar, btnExcluir e btnExibir, com Text como os que vemos na figura a seguir:



Na segunda TabPage, inclua um DataGridView (Coleção Data). Mude seu nome para dgvLivro e aumente esse componente para que fique à direita e ao fundo do TabPage2 e ancore-o nesses lados, conforme vemos na figura abaixo:



Configure a propriedade `ReadOnly` do `dgvLivros` para `true`, de forma que o usuário não possa digitar valores diretamente nesse controle.

Com isso temos a nossa interface básica de livros pronta para ser usada. Fazemos isso acessando os dados da base Sql Server e fazendo uma chamada a nossa camada de negócio (BLL) que, por sua vez, chama a camada de acesso a dados(DAL) que é responsável por recuperar e persistir informação na base de dados.

Usaremos os eventos dos Botões do formulário para fazer a chamada aos métodos da nossa classe de negócio. Para isso no início do nosso formulário devemos ter as seguintes referências:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Windows.Forms;
using BLL;
using DTO;
```

Note que temos que ter uma referência a camada de negócios BLL a nossa camada DTO - Data Transfer Object, onde definimos a classe Livro.

Vamos começar com o evento Click do btnExibir que irá exibir os dados na TabPage Lista em um DataGridView;

```
private void btnExibir_Click(object sender, EventArgs e)
{
    try
    {
        var bll = new LivroBLL();
        dgvLivro.DataSource = bll.SelecionarLivros();
        dgvLivro.Columns[0].HeaderText = "Identificação";
        dgvLivro.Columns[1].HeaderText = "Código";
        dgvLivro.Columns[2].HeaderText = "Título";
        dgvLivro.Columns[3].HeaderText = "Autor(es)";
    }
    catch (Exception ex)
    {
```

```

        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}

```

Nesse código criamos uma instância da classe LivroBLL() e em seguida usamos o método SelecionarLivros() que irá retornar um DataTable e exibir os produtos no DataGridView, através da sua propriedade DataSource. Assim, essa propriedade informa o conjunto de dados que será exibido. O método SelecionarLivros() retorna um DataTable, que é o componente que armazena o resultado de um comando Select aplicado no Banco de Dados. Em seguida, configuramos os cabeçalhos das colunas do DataGridView para que seja mais fácil identificar cada coluna. Sem isso, os nomes dos campos é que seriam exibidos como cabeçalhos; isso, além de dificultar o entendimento, torna menos seguro o nosso ambiente de banco de dados, uma vez que revelaríamos ao usuário final os nomes dos campos, o que facilitaria algum ataque de Sql Injection.

Vejamos agora o código do evento Click do botão btnNovo:

```

private void btnNovo_Click(object sender, EventArgs e)
{
    var livro = new Livro(0, "", "", "");
    livro.CodigoLivro = txtCodigoLivro.Text;
    livro.TituloLivro = txtTituloLivro.Text;
    livro.AutorLivro = txtAutorLivro.Text;
    try
    {
        var bll = new LivroBLL();
        bll.NovoLivro(livro);
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}

```

Primeiro criamos uma instância vazia da classe Livro e definimos as propriedades CódigoLivro, TituloLivro e AutorLivro, atribuindo a elas os valores fornecidos nos campos digitados pelo usuário através das caixas de texto txtCodigoLivro, txtTituloLivro e txtAutorLivro. Não precisamos informar o código do livro (idLivro) pois o mesmo é controlado pelo SGBD visto que definimos este campo como do tipo identity.

A seguir criamos uma instância da classe LivroBLL() no namespace BLL e em seguida usamos o método NovoLivro (livro) para incluir um novo livro na base de dados. Os dados desse novo livro estão armazenados no objeto livro que criamos a partir da classe Livro da pasta DTO de nosso projeto. Observe que passamos como parâmetro um objeto Livro e não valores escalares.

Logo abaixo temos o código associado ao evento Click do botão Alterar:

```

private void btnAlterar_Click(object sender, EventArgs e)
{
    var livro = new Livro(int.Parse(txtIdLivro.Text),
                          txtCodigoLivro.Text,
                          txtTituloLivro.Text,
                          txtAutorLivro.Text);
    try
    {
        var bll = new LivroBLL();
        bll.AlterarLivro(livro);
    }
    catch (Exception ex)
    {

```

```

        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}

```

Primeiro criamos uma instância vazia da classe Livro e definimos as propriedades CódigoLivro, TítuloLivro e AutorLivro, atribuindo a elas os valores fornecidos nos campos digitados pelo usuário através das caixas de texto txtCodigoLivro, txtTituloLivro e txtAutorLivro.

Aqui precisamos informar o ID do livro para identificar o livro a ser alterado; esse livro será buscado no Banco de Dados através da sua chave primária, o idLivro.

A seguir criamos uma instância da classe LivroBLL() no namespace BLL e em seguida usamos o método AlterarLivro(livro) para alterar um livro na base de dados. Observe que passamos como parâmetro um objeto livro e não valores escalares. O método AlterarLivro recebe esse objeto livro como parâmetro, cria um objeto LivroDAL para acessar o banco de dados e que utilizará o idLivro como chave de busca no comando Select na tabela Livro.

Vejamos agora o código do evento Click do botão Excluir:

```

private void btnExcluir_Click(object sender, EventArgs e)
{
    var livro = new Livro(Convert.ToInt32(txtIdLivro.Text), "", "", "");
    try
    {
        var bll = new LivroBLL();
        bll.ExcluirLivro(livro);
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}

```

Primeiro criamos uma instância da classe Livro e definimos a propriedade idLivro atribuindo a ela o valor fornecido pelo usuário através da caixa de texto txtIdLivro. Esse valor já deverá estar preenchido, pois o usuário não pode digitar nesse campo, a não ser no botão de pesquisa (btnProcurar), que ainda programaremos.

A seguir criamos uma instância da classe LivroBLL() no namespace BLL e em seguida usamos o método ExcluirLivro(livro) para excluir um livro na base de dados, identificando esse livro pelo atributo idLivro do parâmetro livro passado para o método. Observe que, nesse caso, não precisamos preencher os demais campos (código, título e autor); Finalmente temos o código do botão Procurar:

```

private void btnProcurar_Click(object sender, EventArgs e)
{
    string codigo = txtCodigo.Text;
    var livro = new Livro(0, codigo, "", "");
    try
    {
        var bll = new LivroBLL();
        livro = bll.SelecionarLivroPorCodigo(codigo);
        txtIdLivro.Text = livro.IdLivro.ToString();
        txtCodigoLivro.Text = livro.CódigoLivro;
        txtTituloLivro.Text = livro.TítuloLivro;
        txtAutorLivro.Text = livro.AutorLivro;
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}

```

```
    }  
}
```

Neste código obtemos a chave primária do livro (IdLivro) a partir da caixa de texto txtIdLivro.Text. A seguir criamos uma instância da classe Livro() e invocamos o método SelecionarLivroPorID usando o id obtido; por fim exibimos os dados do livro no formulário.

Com isso encerramos o nosso formulário de livros, que acessa o SQL Server usando uma arquitetura em camadas. Aplicamos os conceitos básicos da orientação a objetos e de bancos de dados.

Projeto final

Nas BLLs, especifique regras de negócio para exclusão de livros e de leitores e as implemente. Especifique nas BLLs também as regras que estabelecem quando um livro pode ser emprestado e quando um leitor pode emprestar livros, com limite de no máximo 5 livros por leitor em cada momento.

Crie o formulário de manutenção de leitores, usando os conceitos discutidos no formulário de manutenção de livros.

Crie um formulário para empréstimo e devolução de livros.

Consultório Médico - finalização

Na próxima parte de nosso artigo, iremos finalizar a padronização de nosso sistema alterando o formulário principal e iremos nos aprofundar nas características principais da Programação Orientada à Objetos.

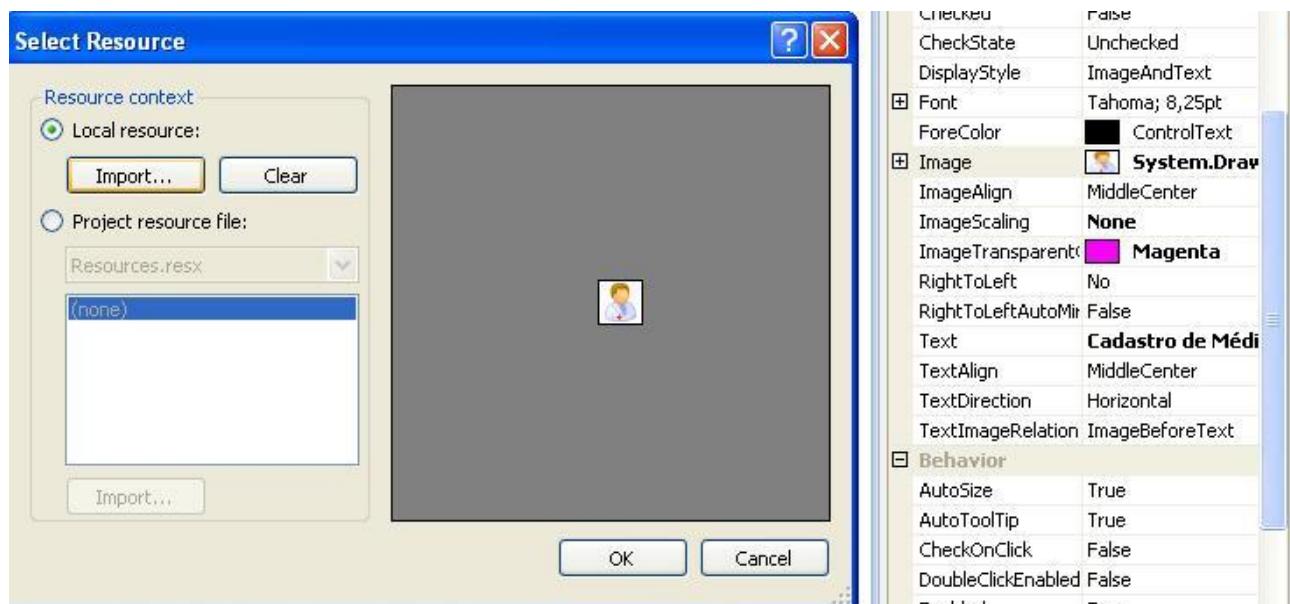
Assim conseguiremos projetar um sistema melhorado, com alta performance simulando um sistema comercial para pequenas e médias empresas.

Parte 3

Olá pessoal, continuamos nossa série de artigos criando aplicações simples em Windows Forms usando a linguagem C# e o banco de dados SQL Server.

Nesta parte iremos finalizar nossa padronização no sistema aplicando algumas configurações personalizadas ao formulário principal. Falaremos também sobre os conceitos de **Programação Orientada à Objetos**. Acompanhem!

De início, vamos alterar as imagens dos menus do form principal. Para isso, abra o form, clique nos botões do menu, se não tiver dado nome a eles, aproveite e já dê o nome de Médicos, Pacientes e Consultas. Agora pesquise no Google Imagens as respectivas imagens para cada botão, salve em seu pc, clique em cada botão e, no menu Propriedades, clique no botão ao lado do atributo **Image** e clique em **Import**, como mostra a imagem abaixo:



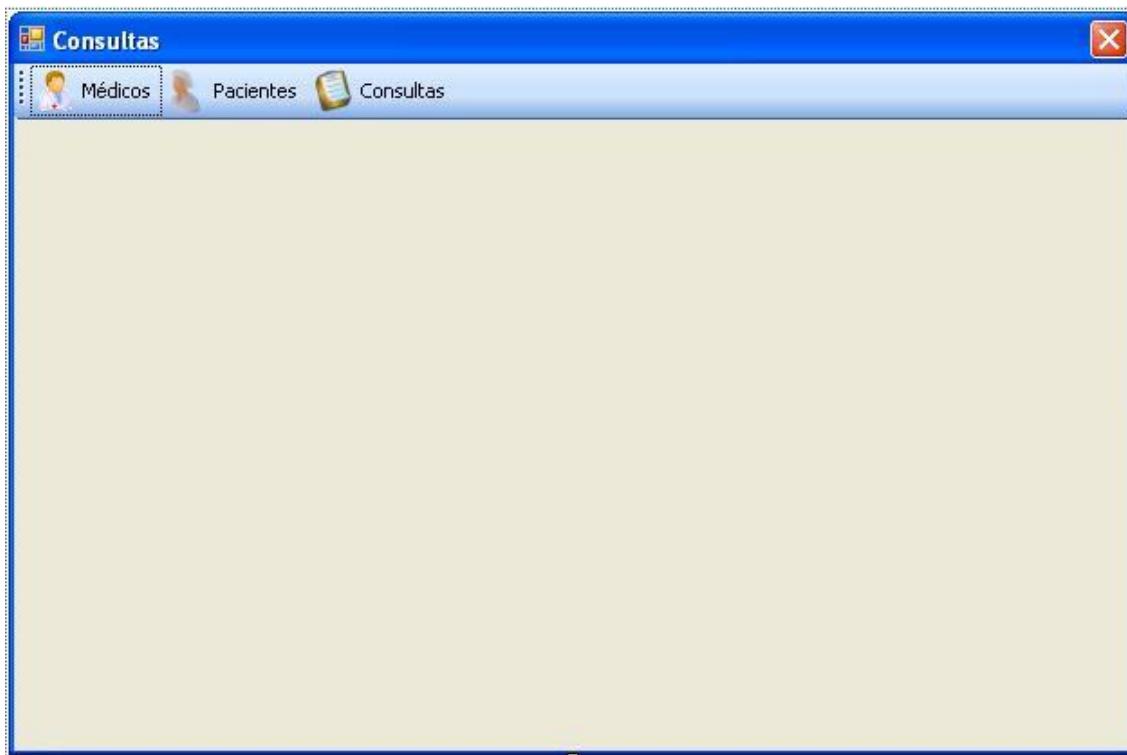
Use de preferência imagens com tamanho máximo de 48x48.

Após inserir as imagens clique no atributo **ImageScaling** e o deixe como None para a imagem ficar com o tamanho real. Se a imagem não ficar transparente, é só alterar a propriedade **ImageTransparentColor** para a cor da sua imagem.

Podemos alterar nosso menu de navegação para que fique, por exemplo, à esquerda de nossa janela e não em cima. Por meio da propriedade **Dock** podemos fazer isso, como mostra a imagem abaixo:

Ok, no meu exemplo vou deixá-lo no topo mesmo, mas se preferir pode alterar a propriedade **Dock** como mostrou a imagem.

A princípio, nosso formulário está pronto. Com cadastros e funcionalidades simples, nos deu uma boa visão de como é fácil trabalhar com os controles do Visual Studio.



Agora imagine o seguinte cenário: uma aplicação dessa feita para a vida real onde, no cadastro pacientes, não são 5 nem 10 nem 100 e sim **1000 registros**.

Como ficaria se precisarmos navegar no registro 999, por exemplo? Iriamos de um em um? Ficaria complicado, concordam? Por isso digo que até o momento montamos um sistema o mais simples possível.

A partir de agora iremos mudar nosso foco e simular uma aplicação de verdade, para um potencial cliente, utilizando alguns conceitos da **Programação Orientada à Objetos**, como a **Herança**.

De forma simplista o conceito de **Herança** significa que uma ou mais classes filhas herdam atributos e métodos da classe pai (conhecida também como classe base). A herança é usada com a intenção de reaproveitar o código e assim garantir uma alta produtividade em nosso sistema.

Vamos trabalhar também em nosso sistema com o conceito de **Parametrização**, que é o conceito de implementar detalhes e parâmetros ao sistema de acordo com as necessidades do cliente. Seguindo este conceito, não teremos mais um formulário como o de Pacientes, com os botões avançar e retroceder os registros, pois quando abrirmos o form de Pacientes, não queremos ver todos os Pacientes e sim ver determinado Paciente.

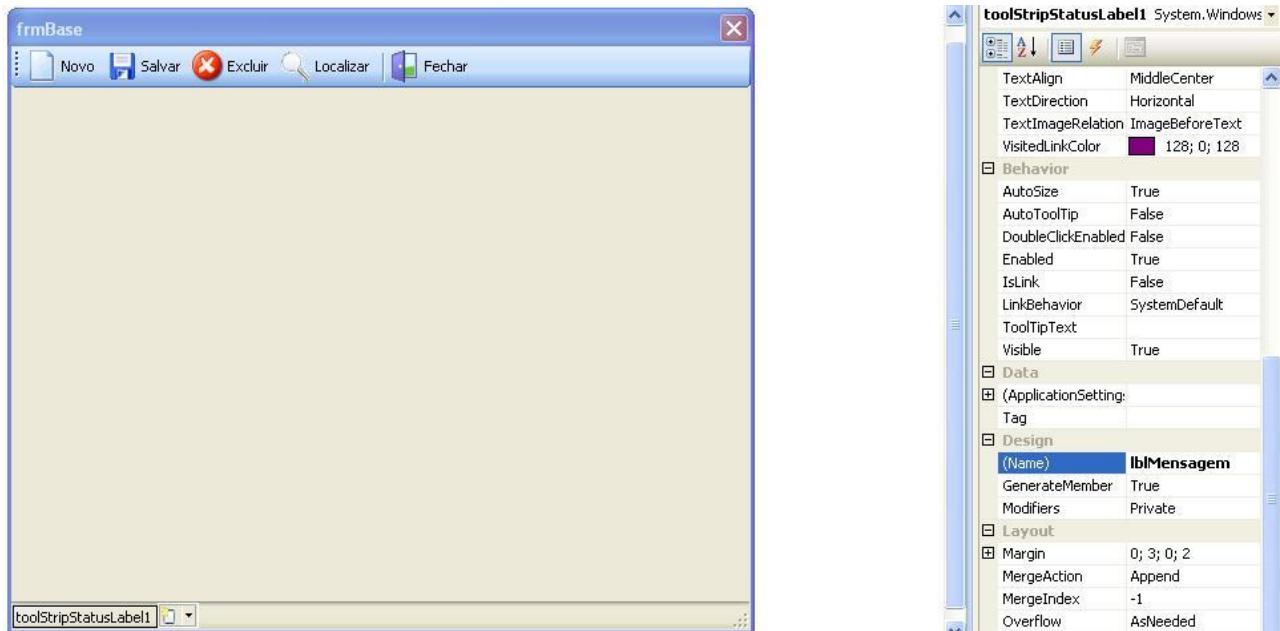
Será feito assim em todos os formulários, por meio do conceito de Herança. Teremos também um formulário de pesquisa. Comecemos então nosso formulário base principal que será o “formulário pai”:

Clique em **Add > Windows Form**, dê o nome de **frmBase** e clique em **OK**.

Será gerado nosso formulário, clique nele, abra a janela Properties (F4) e altere as seguintes propriedades:

- **Start Position** – coloque CenterScreen para que o form abra no meio da tela
- **MaximizeBox** – false
- **MinimizeBox** – false – para que só apareça o botão de fechar no form
- **KeyPreview** – true, para ativar o uso do teclado nos eventos do form
- **FormBorderStyle** – altere para FixedDialog para que o form não possa ser redimensionado

Agora vamos inserir os controles que serão padrões nos demais forms. Arraste uma **ToolStrip** de nossa Toolbox para o form e insira 4 botões, 1 separador e mais 1 botão, nessa ordem referentes aos botões **Novo**, **Salvar**, **Excluir**, **Localizar** e **Fechar** conforme mostra a imagem abaixo:



Altere a propriedade **DisplayStyle** para **Image and Text** dos botões, a propriedade **ImageScaling** para **None** e insira imagens para os botões referentes como fizemos no form anterior. Altere também a propriedade **Design** dos botões para podermos identificá-los mais facilmente quando formos usá-los na programação.

Na ordem altere para **btnNovo**, **btnSalvar**, **btnExcluir**, **btnLocalizar** e **btnFechar**.

Adicione o controle **StatusStrip** a seu form, nele adicione um **StatusStripLabel**, dê o nome de **lbtMensagem** e deixe a propriedade **Text** em branco para que possamos configurar via código uma mensagem ao usuário quando ele realizar determinada ação.

Clique com o botão direito no seu form e clique em View Code ou simplesmente aperte **F7** para ir a página de códigos. Nela, vamos criar um **enumerador**, com os itens Inserindo, Navegando e Editando, e criaremos uma variável privada deste enumerador para sabermos qual é o status do sistema, se por exemplo o usuário estiver inserindo, devemos desabilitar o botão de excluir, ou quando estiver localizando, devemos desabilitar o botão de salvar.

Desta forma estaremos otimizando nosso código e evitando erros do usuário. A imagem abaixo mostra nosso código:

```

10  namespace Consultas
11  {
12      public partial class frmBase : Form
13      {
14          public enum StatusCadastro
15          {
16              scInserindo,
17              scNavegando,
18              scEditando
19          }
20
21          private StatusCadastro sStatus;
22
23          public frmBase()
24          {
25              InitializeComponent();
26          }
27
28      }

```

Vamos voltar ao nosso form, no modo visual e implementar o código para o botão Fechar, como parte de nosso exemplo. Primeiro dê dois cliques em cima do botão de fechar e chame o método **Close()**. Depois, volte ao form, abra a janela de propriedades do mesmo, clique nos Eventos, selecione o evento **KeyDown**, dê dois cliques nele e insira o código abaixo para que o form se feche ao apertar ESC.

```

28     private void btnFechar_Click(object sender, EventArgs e)
29     {
30         Close();
31     }
32
33     private void frmBase_KeyDown(object sender, KeyEventArgs e)
34     {
35         if (e.KeyCode == Keys.Escape)
36         {
37             Close();|
38         }
39     }

```

Usando esse conceito de Herança em nosso sistema, só utilizaremos este código acima apenas uma vez e os demais herdarão as funcionalidades do formulário base, o que melhora nosso desempenho e temos facilidade se precisarmos dar manutenção.

Na próxima parte de nosso artigo, iremos continuar a codificação de nosso formulário base implementando os métodos que iremos usar na herança do mesmo.

Parte 4

Nesta parte vamos continuar com a codificação de nosso formulário base implementando os métodos que iremos usar na herança visual dos demais.

Devemos ter a preocupação de, depois que o usuário gravar um registro por exemplo, os controles usados, como **combobox**, **textbox** e outros, sejam limpados automaticamente. Para isso devemos criar um método que limpa os campos após a gravação de um registro qualquer.

Abra o **frmBase.cs** e crie um novo método do tipo **private** com o nome **LimpaControles** que não irá nos retornar nada, como mostra a imagem abaixo:

```

41     private void LimpaControles()
42     {
43         foreach (Control ctr in this.Controls)
44         {
45             if (ctr is TextBox)
46                 (ctr as TextBox).Text = "";
47
48             if (ctr is ComboBox)
49                 (ctr as ComboBox).SelectedIndex = -1;
50
51             if (ctr is ListBox)
52                 (ctr as ListBox).SelectedIndex = -1;
53
54             if (ctr is RadioButton)
55                 (ctr as RadioButton).Checked = false;
56
57             if (ctr is CheckBox)
58                 (ctr as CheckBox).Checked = false;
59
60             if (ctr is CheckedListBox)
61             {
62                 foreach (ListControl item in (ctr as CheckedListBox).Items)
63                     item.SelectedIndex = -1;|
64             }
65         }
66     }
67 }
68

```

Acima fiz um laço do tipo **foreach** na classe nativa **Control** (pertencente ao namespace Windows.Forms) com o objetivo de ser feita uma “varredura” em meu formulário, percorrendo todos os controles existentes dentro do meu form. A primeira verificação, por exemplo, foi pra saber se existe um textbox em meu form e, se existir, ele será limpado. E assim faço com os demais. Lembrando que em alguns controles, como **Label** e **Button**, não tem necessidade de serem limpos. Conforme você for precisando adicionar ou substituir um controle por outro é só adicionar o tipo dele em nosso **foreach**.

Se precisarmos fazer o mesmo a um controle do tipo **CheckedListBox**, devemos fazer outro foreach, como mostrou nosso último **if** da imagem acima.

Continuando com nosso conceito de trabalhar com Herança, iremos criar os métodos **Salvar**, **Excluir** e **Localizar** apenas neste formulário base, e iremos fazer a chamada desses métodos nos demais formulários, poupando assim o retrabalho de ter que digitar o mesmo método diversas vezes. Iremos começar criando o método Salvar.

```

68     public virtual bool Salvar()
69     {
70         return false;
71     }
72
73     public virtual bool Excluir()
74     {
75         return false;
76     }
77
78     public virtual bool Localizar()
79     {
80         return false;
81     }

```

Acima criei o método do tipo booleano passando como retorno o valor **false**, ele só me retornará **true** quando o registro for salvo no banco. Faço o mesmo com os métodos Excluir e Localizar.

Agora vá ao modo design do formulário base e dê dois cliques nos botões Salvar e Excluir. Dentro deles insira o seguinte código:

```

83     private void btnSalvar_Click(object sender, EventArgs e)
84     {
85         if (Salvar())
86         {
87             LimpaControles();
88             MessageBox.Show("Registro salvo com sucesso", "Aviso do Sistema",
89                             MessageBoxButtons.OK, MessageBoxIcon.Information);
90         }
91         else
92         {
93             MessageBox.Show("O registro não foi salvo, por favor verifique os erros!", "Erro",
94                             MessageBoxButtons.OK, MessageBoxIcon.Error);
95         }
96     }
97
98     private void btnExcluir_Click(object sender, EventArgs e)
99     {
100        if (Excluir())
101        {
102            LimpaControles();
103            MessageBox.Show("Registro excluido com sucesso", "Aviso do Sistema",
104                            MessageBoxButtons.OK, MessageBoxIcon.Information);
105        }
106        else
107        {
108            MessageBox.Show("O registro não foi excluido, por favor verifique os erros!", "Erro",
109                            MessageBoxButtons.OK, MessageBoxIcon.Error);
110        }
111    }

```

O que fiz foi chamar os métodos referentes aos respectivos botões e dependendo da resposta do método entro no meu if ou no meu else. Em minhas verificações coloquei o método LimpaControles, pois ele irá entrar verificar se o registro foi gravado, por exemplo, se foi ele limpa os controles e exibe a mensagem ao usuário, ou que o registro foi gravado ou que deu erro, isso dependendo do que acontecer em nosso sistema.

Uma característica que será implementada é a de que quando forem abertos os formulários sempre virão registros vazios, ou seja, se desejar um registro específico, o usuário terá que localizar o mesmo por meio do respectivo botão. Depois de salvo um registro, por exemplo, eles são automaticamente limpados.

Vamos agora implementar o **enum** que criamos no artigo anterior, em nosso botão Salvar. Se o usuário salvou o registro ou excluiu, devemos configurar nosso enum para **Navegando**, como mostra a imagem a seguir:

```
if (Salvar())
{
    sStatus = StatusCadastro.scNavegando;
    LimpaControles();
    MessageBox.Show("Registro salvo com sucesso", "Aviso do Sistema",
                    MessageBoxButtons.OK, MessageBoxIcon.Information);
}
```

Devemos também dentro desses métodos fazer a chamada a outro que iremos criar para que os controles sejam habilitados ou desabilitados.

Agora vamos criar os métodos para os botões Novo e Localizar que são iguais com os dos botões Salvar e Excluir, como você pode perceber na imagem:

```
115     private void btnNovo_Click(object sender, EventArgs e)
116     {
117         sStatus = StatusCadastro.scInserindo;
118         LimpaControles();
119     }
120
121     private void btnLocalizar_Click(object sender, EventArgs e)
122     {
123         if (Localizar())
124         {
125             sStatus = StatusCadastro.scEditando;
126         }
127     }
```

Lembrando que após criarmos nosso método que habilita/desabilita os controles do form devemos implementá-lo aos botões Novo e Localizar.

Vamos criar um método que será implementado nos formulários que herdarão de meu formulário base, que será o CarregaValores, a função dele será a de retornar os dados parametrizados do banco e carregar meus controles, vou dizer que, por exemplo, o campo **txtNome** irá vir carregado com a coluna de nomes do meu banco e assim por diante. Faça a chamada a este método no botão Localizar.

Crie um novo método do tipo public virtual como os anteriores só que ele será do tipo void, ou seja, não terá retorno algum. Crie também o método para habilitar / desabilitar os controles no form, como a imagem abaixo nos mostra:

```

private void HabilitaDesabilitaControles(bool bValue)
{
    //percorre os controles da tela e os habilita ou desabilita
    foreach (Control ctr in this.Controls)
    {
        if (ctr is ToolStrip)
            continue;

        ctr.Enabled = bValue;
    }
}

```

O que fiz foi um foreach, que percorre todos os meus controles, parecido com os do método LimpaControles. Só que aqui ele faz um if pra verificar se o controle é um ToolStrip, se for ele continua, se não for, ele habilita nossa variável bValue que foi passada como parâmetro em nosso método.

Agora dentro deste mesmo método, faço as verificações em meus botões para habilitá-los ou desabilitá-los.

```

private void HabilitaDesabilitaControles(bool bValue)
{
    //percorre os controles da tela e os habilita ou desabilita
    foreach (Control ctr in this.Controls)
    {
        if (ctr is ToolStrip)
            continue;

        ctr.Enabled = bValue;
    }

    //habilitar os botões

    //Novo - será habilitado somente quando for navegação
    btnNovo.Enabled = (sStatus == StatusCadastro.scNavegando);

    //Salvar
    btnSalvar.Enabled = (sStatus == StatusCadastro.scNavegando || 
                         sStatus == StatusCadastro.scInserindo);

    //Excluir
    btnExcluir.Enabled = (sStatus == StatusCadastro.scEditando);

    //Localizar
    btnLocalizar.Enabled = (sStatus == StatusCadastro.scNavegando);

    //Fechar
    btnFechar.Enabled = true;
}

```

Acima fiz o seguinte: passei o status de cada botão, por exemplo, no primeiro botão, o Novo, atribui à ele que o status será o Navegando e assim fiz com os demais. No fim atribui que o botão Fechar sempre estará habilitado. Assim, dependendo da ação do usuário o botão específico estará habilitado ou desabilitado (com exceção do Fechar que terá valor fixo).

Agora é só aplicar os métodos aos respectivos botões passando os valores, true ou false, dependendo de cada situação. Por exemplo, no botão novo, passo meu método com o valor true, porque quero habilitar os controles para que o usuário insira um novo registro, já no botão excluir, passo o valor false ao meu método porque já exclui, não preciso mais que os controles sejam habilitados e assim sucessivamente. Abaixo um exemplo do botão Localizar:

```
private void btnLocalizar_Click(object sender, EventArgs e)
{
    if (Localizar())
    {
        sStatus = StatusCadastro.scEditando;
        HabilitaDesabilitaControles(true);
        CarregaValores();
    }
}
```

Finalizando, vá ao modo **Design** de nosso form e dê dois cliques no form para ir ao evento Load do mesmo. Nele, insira os seguintes códigos:

```
private void frmBase_Load(object sender, EventArgs e)
{
    sStatus = StatusCadastro.scNavegando;
    LimpaControles();
    HabilitaDesabilitaControles(false);
}
```

Fazendo isso, toda vez que o form for carregado ele iniciará com o status Navegando, com os controles limpos e com o método **HabilitaDesabilitaControles** como **false**. Assim inicializamos os métodos em nosso form de uma forma prática e sem erros, permitindo ao usuário praticidade e performance na hora de navegar nos formulários.

Na próxima parte de nosso artigo, iremos criar nosso formulário base de Pesquisa, com controles específicos para que possamos realizar a pesquisa no banco de dados.

Parte 5

Nesta parte vamos montar nossa tela de pesquisa, que será parametrizada, ou seja, nos retornará apenas o que pesquisarmos e não toda a nossa base de dados, o que em uma aplicação real nos traria muita dor de cabeça se não fosse usado o recurso de parametrização que estamos implementando. Nossa tela de pesquisa será também genérica. Montaremos a tela padrão e, a partir dela, serão montadas as telas herdadas que forem necessárias no decorrer de nossos artigos. Com isso ganharemos e muito em performance em nosso sistema.

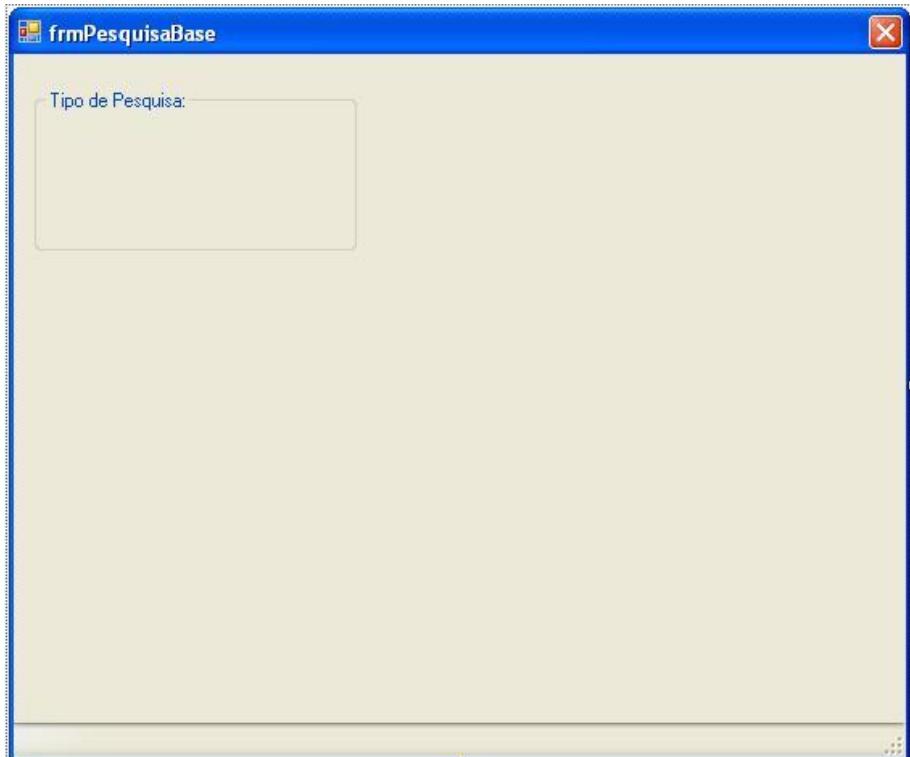
Abra sua solution e clique em **Add > New Windows Form**. Dê a ele o nome de **frmPesquisaBase**, pois este será o nosso formulário base de Pesquisa.

Será gerado nosso formulário, clique nele, abra a janela Properties (F4) e altere as seguintes propriedades:

- **Start Position** – coloque CenterScreen para que o form abra no meio da tela
- **MaximizeBox** – false
- **MinimizeBox** – false – para que só apareça o botão de fechar no form
- **KeyPreview** – true, para ativar o uso do teclado nos eventos do form
- **FormBorderStyle** – altere para FixedSingle (faça o mesmo nas propriedades do formulário Base)
- **ShowInTaskbar** – false, para não mostrarmos o form no taskbar

Adicione o controle **StatusStrip** a seu form (da mesma forma como fizemos no formulário base) , nele adicione um **StatusStripLabel**, dê o nome de **lblMensagem** e deixe a propriedade **Text** em branco para que possamos mostrar ao usuário qual foi a quantidade de registros retornados após ele ter feito uma pesquisa.

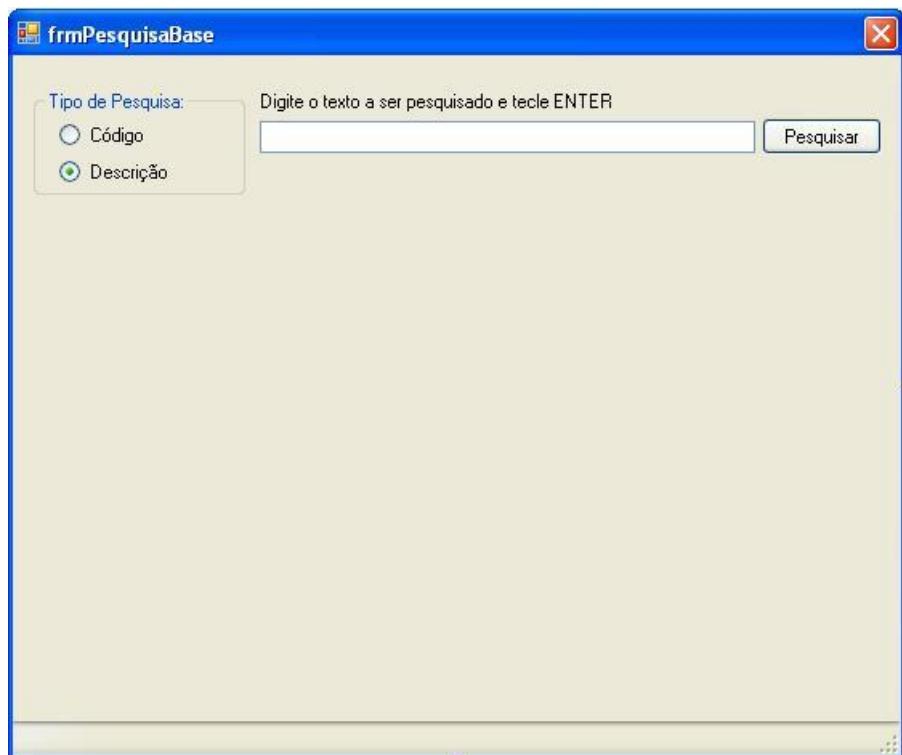
Nosso formulário terá duas formas de pesquisar: ou pelo código do usuário, ou pela descrição. Vamos adicionar um **GroupBox** ao nosso form e dar o nome de **Tipo de Pesquisa**, como mostra a imagem a seguir:



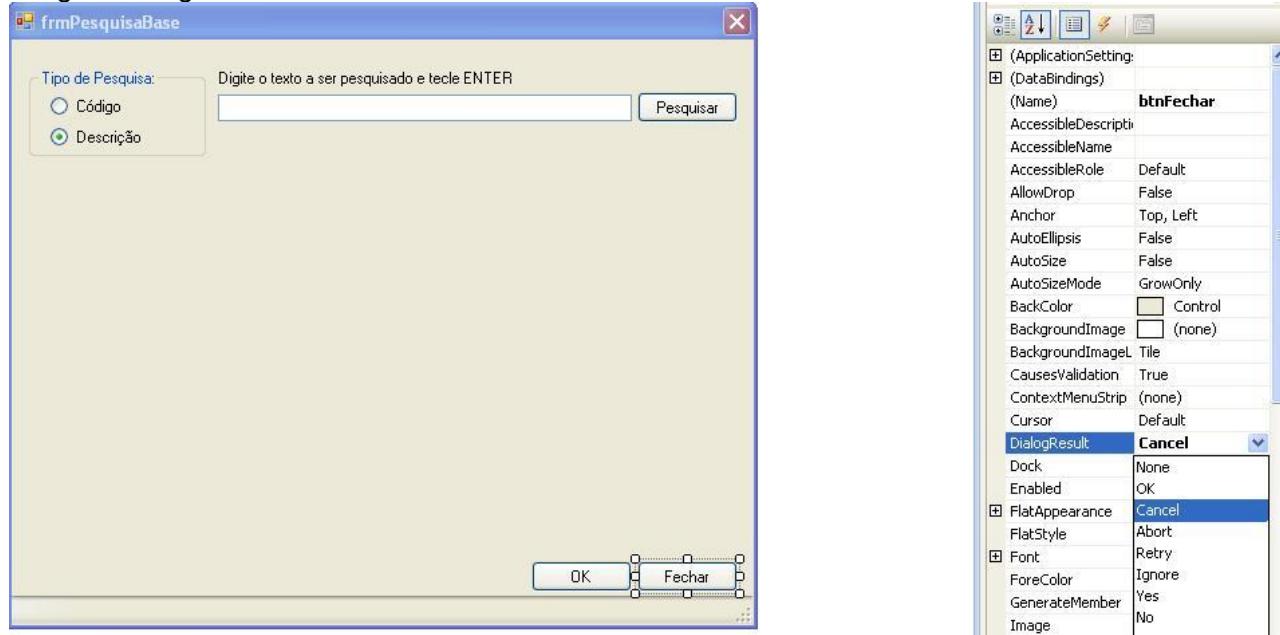
Agora dentro dele adicione dois **RadioButton** com as propriedades **Text Código e Descrição**. Altere o ID de cada um deles para **rbtCodigo** e **rbtDescricao**, respectivamente e deixe a propriedade **Checked** do RadioButton **Descrição** setada para **True**, para que o form sempre inicialize com ele checado, que é o mais comum de se pesquisar.

Adicione também, ao lado do **GroupBox**, um **Label**, um **TextBox** e um **Button**, para que o usuário possa digitar no campo a descrição referente ao usuário.

Altere o ID deles para **lblDescricao**, **txtPesquisar** e **btnPesquisar**. Deixar seu form como o da imagem ao lado:



Agora adicione dois botões no canto inferior direito do seu form, um que é o botão OK e outro o botão Fechar. Dê dois cliques no Fechar e chame o método **Close**. A esses botões sete a propriedade **DialogResult** de cada um como **OK** e **Cancel**, respectivamente, como mostra a imagem a seguir:

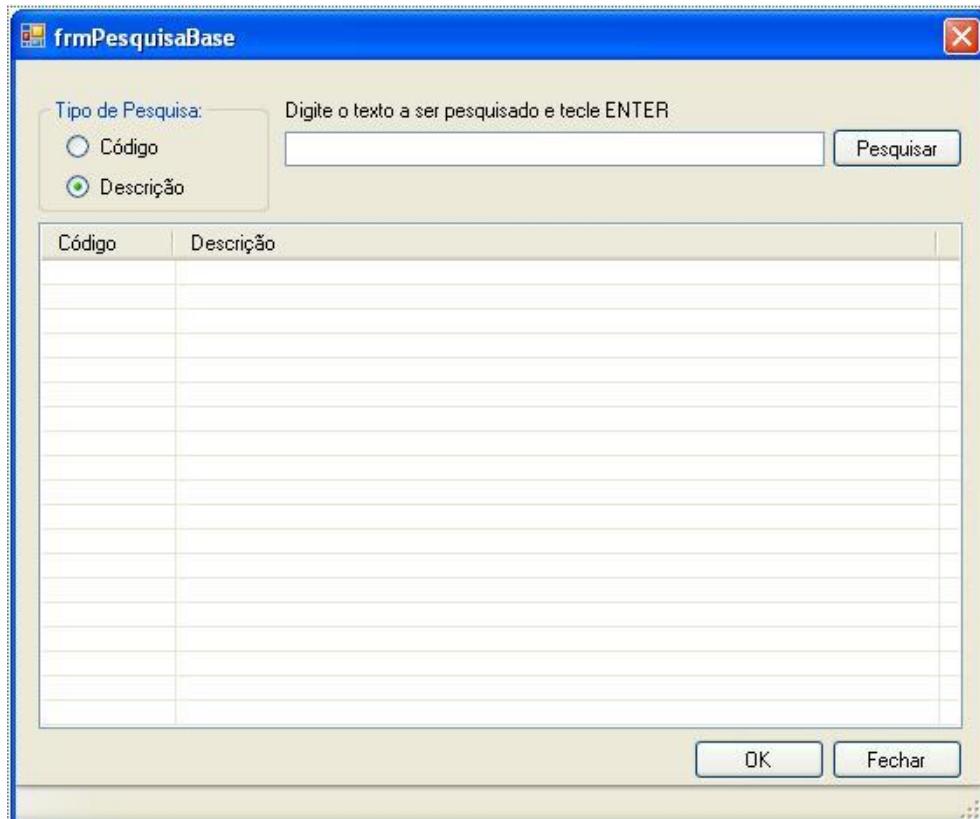


Agora abra o evento **KeyDown** do seu form e coloque aquela verificação para que, quando o usuário pressionar **ESC** ou **ENTER**, o form se feche. Sete o valor do **DialogResult** para **Cancel**, como mostra a imagem abaixo:

```
private void frmPesquisaBase_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape)
    {
        Close();
        DialogResult = DialogResult.Cancel;
    }

    if (e.KeyCode == Keys.Enter)
    {
        Close();
        DialogResult = DialogResult.Cancel;
    }
}
```

Agora vamos implementar um **ListView** ao nosso formulário. Então abra a Toolbox, arraste um controle **ListView** ao form e dê a propriedade **ID** o nome **IstPesquisa**. Ainda nas propriedades, altere a propriedade **View** para **Details** e abra a opção **Columns** para adicionar duas colunas ao nosso **ListView**. Clique em **Add** e altere a propriedade **Text** para **Código**, faça o mesmo na próxima, dê o nome **Descrição** e clique em **OK**. Redimensione sua coluna **Descrição** se desejar. Altere a propriedade **FullRowSelect** para **True** para que, quando o usuário clique no campo **Descrição**, por exemplo, seja selecionado toda a linha e não só o campo clicado. Altere para **True** também a propriedade **GridLines**, para que nosso **ListView** fique com as linhas do Grid. Finalizando o design de nosso **ListView**, altere a propriedade **MultiSelect** pra **False**, porque queremos que seja selecionado apenas uma linha de nosso grid e não várias. Se você alterou as propriedades como descrito, seu **ListView** deve estar parecido com o da imagem a seguir:



Vamos implementar as funcionalidades de nosso ListView entrando na página de códigos do formulário. Vamos criar uma classe pública do tipo string e a inicializaremos com um valor vazio, para que ela seja preenchida em outras partes de nosso código, como quando o usuário escolher um código em nossa Grid e clicar em OK ou mesmo quando ele der dois cliques no Grid. Vejamos o código:

```
//declaramos a variável pública do tipo string
public string sCdCodigo;

public frmPesquisaBase()
{
    InitializeComponent();
    //inicializamos a variável com valor vazio
    sCdCodigo = "";
```

Agora volte ao modo Design do form e dê dois cliques no botão OK. Insira o seguinte código:

```

private void btnOK_Click(object sender, EventArgs e)
{
    //preencho a variável sCdCodigo
    //faço a verificação se ela está vazia
    if (sCdCodigo == "")
    {
        //se estiver vazia e o foco estiver no ListView preenche
        //a variável passando o Código do ListView como parâmetro
        if (lstPesquisa.Focused)
        {
            sCdCodigo = lstPesquisa.SelectedItems[0].Text;
        }
    }
}

private void lstPesquisa_SelectedIndexChanged(object sender, EventArgs e)
{
    try
    {
        sCdCodigo = lstPesquisa.SelectedItems[0].Text;
    }
    catch
    {

    }
}

```

O que fizemos no botão OK tivemos que fazer também no evento **SelectedIndexChanged** do ListView só que usamos um **Try/Catch** para tratar um erro que não é da aplicação e sim do próprio controle. No evento **DoubleClick** do ListView devemos usar o código para preencher nossa variável e adicionarmos o método **Close** e o **DialogResult** com o valor **OK**, como fizemos no evento **KeyDown** do formulário anteriormente. A imagem a seguir mostra nosso método:

```

private void lstPesquisa_DoubleClick(object sender, EventArgs e)
{
    try
    {
        sCdCodigo = lstPesquisa.SelectedItems[0].Text;
        Close();
        DialogResult = DialogResult.OK;
    }
    catch
    {

    }
}

```

Dessa forma, capturamos praticamente todas as opções que o usuário tem no momento em que vai pesquisar. Seja no clique duplo no resultado da pesquisa, seja no botão OK, seja quando o foco está na pesquisa do usuário.

Na próxima parte de nossa série de artigos, faremos os métodos necessários para que o formulário de pesquisa funcione corretamente. Implementaremos os métodos para que o ListView nos retorne o que o usuário digitar no campo de Pesquisa e iremos customizar nosso DataSet já criado nos artigos anteriores.

[Parte 6](#)

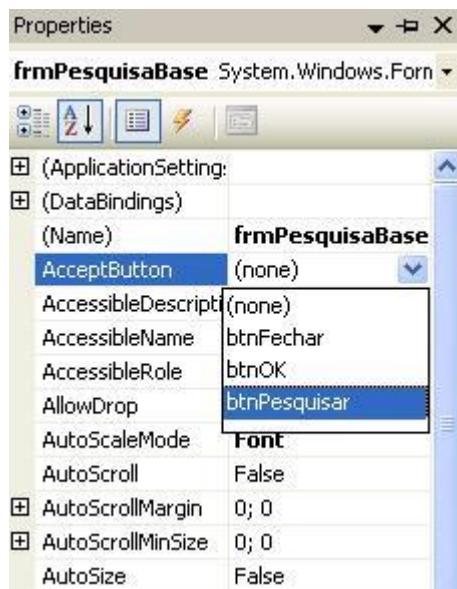
Nesta parte vamos criar os métodos para que o **ListView** nos retorne o que o usuário digitar no campo de **Pesquisa**. Iremos também criar um novo **DataSet** e deixarmos de lado o criado nos artigos anteriores.

Vamos começar criando nosso método de Pesquisa, no arquivo de códigos de nosso formulário de Pesquisa crie um método do tipo **público virtual** e que não irá ter retorno (**void**). Não iremos implementar nada neste método, já que serão implementados apenas nos formulários herdados. Ou seja, em cada formulário de pesquisa terá uma implementação diferente, como a pesquisa de clientes, por exemplo.

```
public virtual void Pesquisar()
{
}
```

Agora vamos ao modo **Design** de nosso formulário. Quero que meu botão de Pesquisa seja chamado quando o usuário digitar algo na caixa de pesquisa e der um **ENTER**. Ou seja, este será o botão padrão de nosso form. Para isso, clicamos em nosso form e, na propriedade **AcceptButton**, escolhemos nosso **btnPesquisar**, como a imagem ao lado nos mostra:

Dê dois cliques no botão Pesquisar para voltarmos à página de códigos de nosso form. No evento **Click** desse botão chame o método **Pesquisar**, faça a configuração de nossa **StatusStrip**, que criamos anteriormente, para retornar ao usuário quantos registros foram encontrados em cada pesquisa que for feita e adicione outras implementações, como mostrado na imagem abaixo:



```
private void btnPesquisar_Click(object sender, EventArgs e)
{
    //chamo o método Pesquisar
    Pesquisar();

    //configuro meu StatusStrip para retornar ao usuário informações referentes à pesquisa
    //para que fique entre aspas a qtde e o registro pesquisado apenas adiciono \'"
    lblMensagem.Text = "Encontrado(s): \"" + lstPesquisa.Items.Count +
        "\" registros com a palavra \"" + txtPesquisa.Text + "\"";

    //aqui habilito meu botão OK somente se minha pesquisa retornar ao menos 1 registro
    btnOK.Enabled = lstPesquisa.Items.Count > 0;

    //faço uma verificação se o botão OK estiver habilitado, meu ListView ganha o foco
    if (btnOK.Enabled)
    {
        lstPesquisa.Focus();
    }
}
```

Além das implementações citadas acima, como você pode ver na imagem, adicionamos mais duas verificações, uma que habilita o botão **OK** de nosso form se a propriedade **Count** for maior que **0**, ou seja, se nosso **ListView** nos retornar ao menos 1 registro em nossa pesquisa, e a outra que, se o botão **OK** estiver habilitado, faz com que nosso **ListView** ganhe o foco. Esta última é necessária porque, assim o usuário pode navegar entre os registros usando as setas do teclado, dando duplo clique para ver determinado registro por exemplo e assim estaremos evitando ao máximo possível erros do usuário.

Agora vamos criar um método que irá carregar o retorno de nossa pesquisa da base de dados em nosso ListView. Crie o método do tipo público void e que terá como parâmetro um DataTable, que será implementado em nossos formulários que herdarão do formulário base. Veja o método a seguir:

```
public void CarregarItens(DataTable dt)
{
    //limpo os registros do ListView
    lstPesquisa.Items.Clear();

    //carrego os dados no ListView
    foreach (DataRow dr in dt.Rows)
    {
        //para cada linha de meu DataTable, insiro uma linha no ListView
        //instancio o ListViewItem, adiciono um item e um subitem, referentes
        //ao código e a descrição que estou pesquisando em meu formulário
        ListViewItem item = new ListViewItem();
        item.Text = dr[0].ToString();
        item.SubItems.Add(dr[1].ToString());

        //aqui adiciono a variável instanciada item
        //carregada com o item e subitem ao ListView
        lstPesquisa.Items.Add(item);
    }
}
```

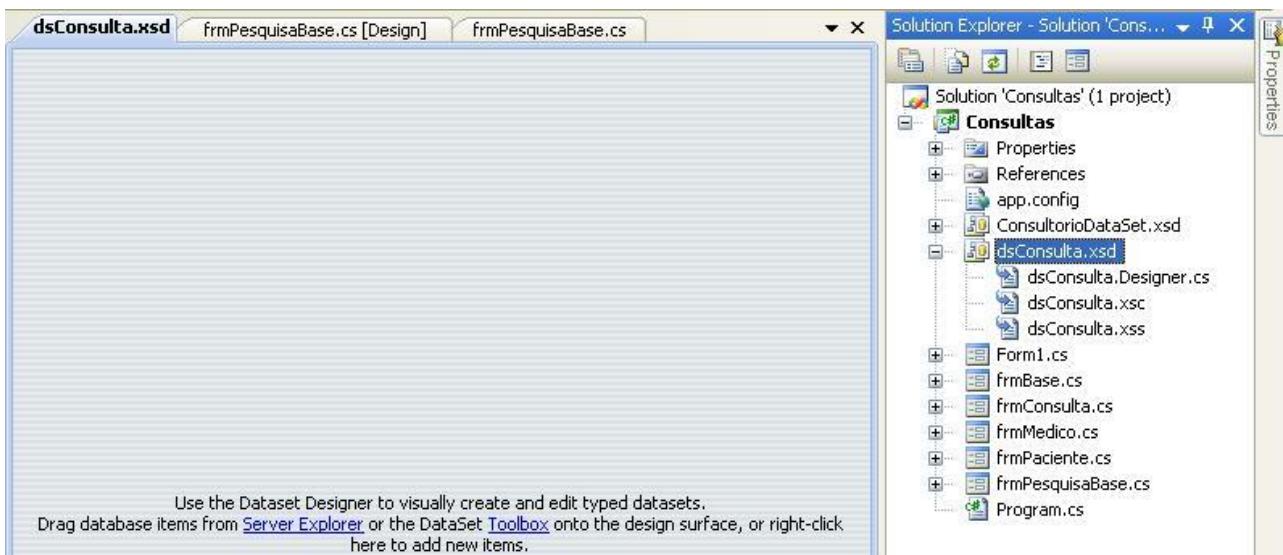
Com este método, finalizamos nossos métodos do formulário de Pesquisa, agora volte ao modo Design, clique nos Radio Buttons **Código** e **Descrição**, vá ao evento **Click**, dê dois cliques e insira o seguinte código:

```
private void rbtCodigo_Click(object sender, EventArgs e)
{
    //quando o usuário clicar no RadioButton, o foco é
    //automaticamente "setado" para o TextBox de Pesquisa
    txtPesquisa.Focus();
}

private void rbtDescricao_Click(object sender, EventArgs e)
{
    txtPesquisa.Focus();
}
```

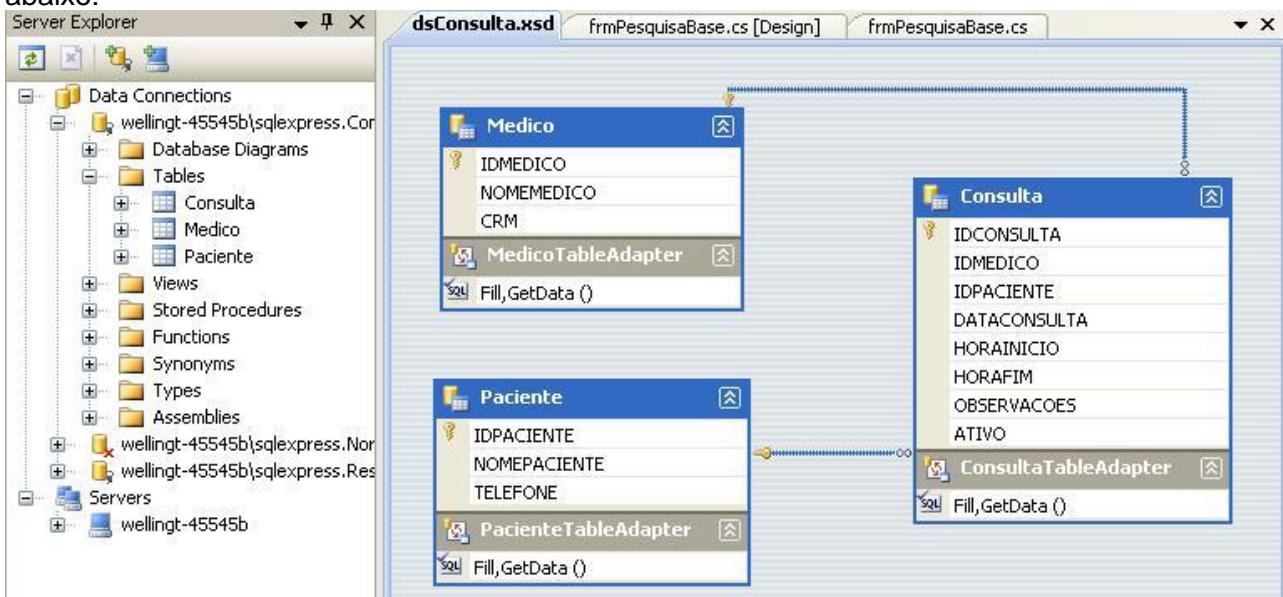
Apenas para que o usuário tenha uma maneira rápida e fácil de pesquisar, melhorando assim a eficiência de seu sistema e a usabilidade do mesmo.

Lembrando que para cada formulário de pesquisa, teremos dois tipos de pesquisa: por código ou por descrição. Com isso em mente, iremos criar um novo **DataSet** e deixaremos de lado o que foi criado em artigos anteriores, pois nele usamos as facilidades do Visual Studio para criá-lo, já nesse estamos usando um outro tipo de abordagem, tendo como base o conceito de **Programação Orientada à Objetos**. Dito isto, abra a **Solution Explorer** (CTRL+W+S), clique em seu projeto com o botão direito e clique em **Add > New Item**. Escolha no menu **Categories** a opção Data, no menu **Templates** escolha DataSet, dê o nome **dsConsulta** e clique em OK.

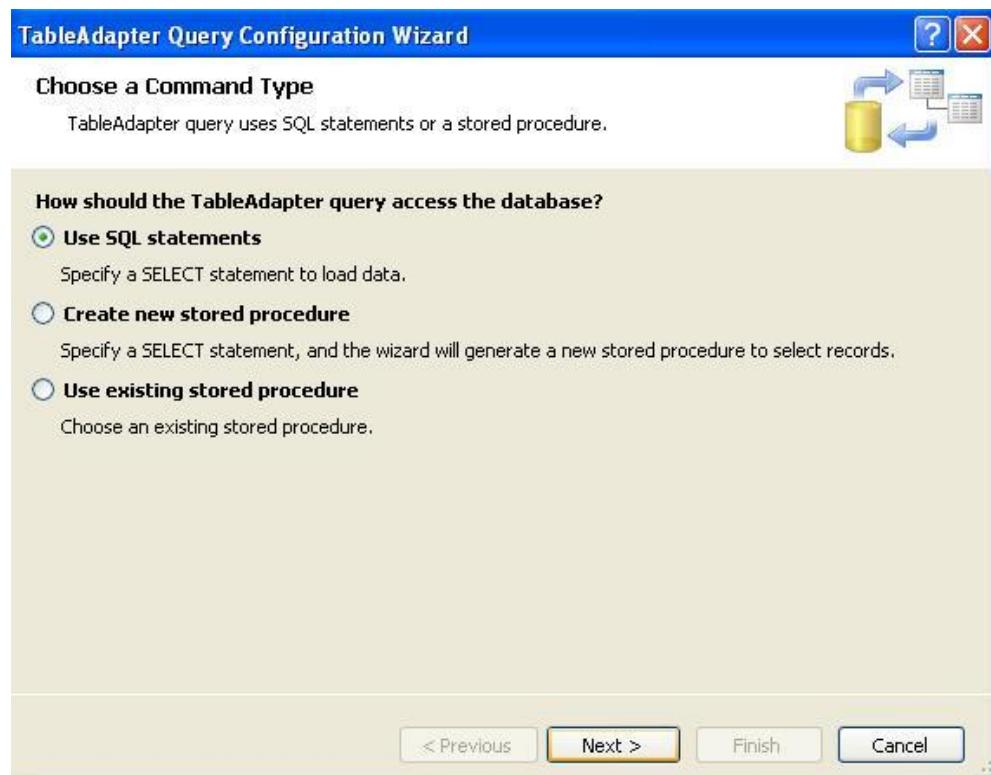


Será um DataSet igual ao nosso criado nas primeiras aulas, a diferença é que desta vez não iremos usar o menu **DataSources** para arrastar as tabelas. Vamos arrastar diretamente do banco por meio do **Server Explorer**, assim ele criará para nós cada **DataTable** e **DataAdapter** referentes às respectivas tabelas para que possamos customizá-las.

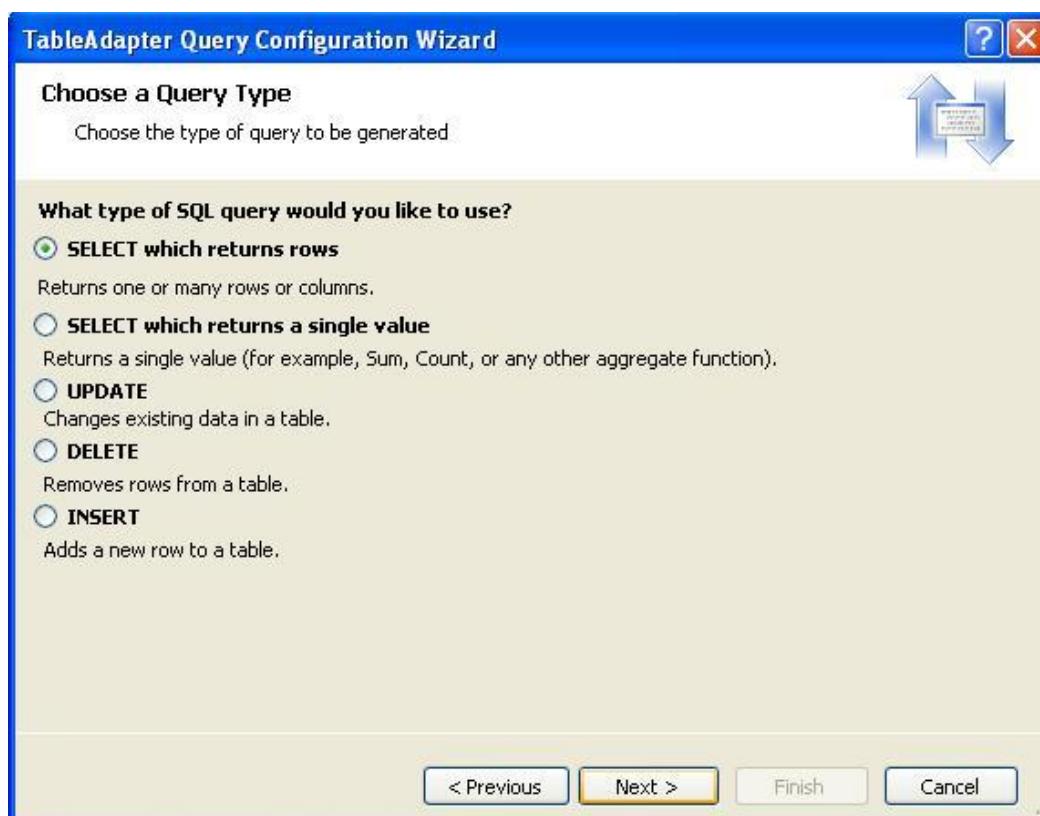
Então abra o **Server Explorer** (CTRL+W+L), expanda seu Database Consultorio, seu menu Tables arraste as tabelas para o DataSet recém criado. Seu DataSet deverá ficar como o da imagem abaixo:



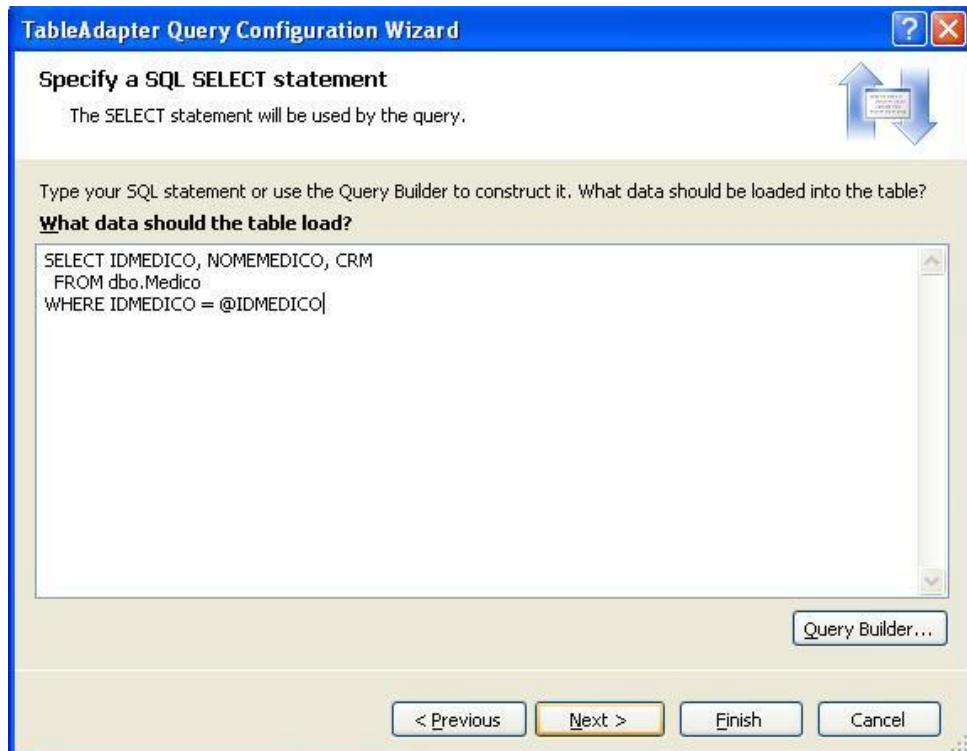
Vamos customizar a tabela de médico para criar mais duas pesquisas, uma por código e outra por nome/descrição. Para isso, clique com o botão direito no **MedicoTableAdapter** da tabela **Medico** e clique em **Add Query**. Irá abrir um wizard que nos perguntará como nossa query irá acessar o banco, existem três opções: por meio de **instruções SQL**, por meio da criação de uma **Stored Procedure** ou por meio do uso de uma Stored Procedure existente. Neste caso, escolha a 1ª opção e clique em Next.



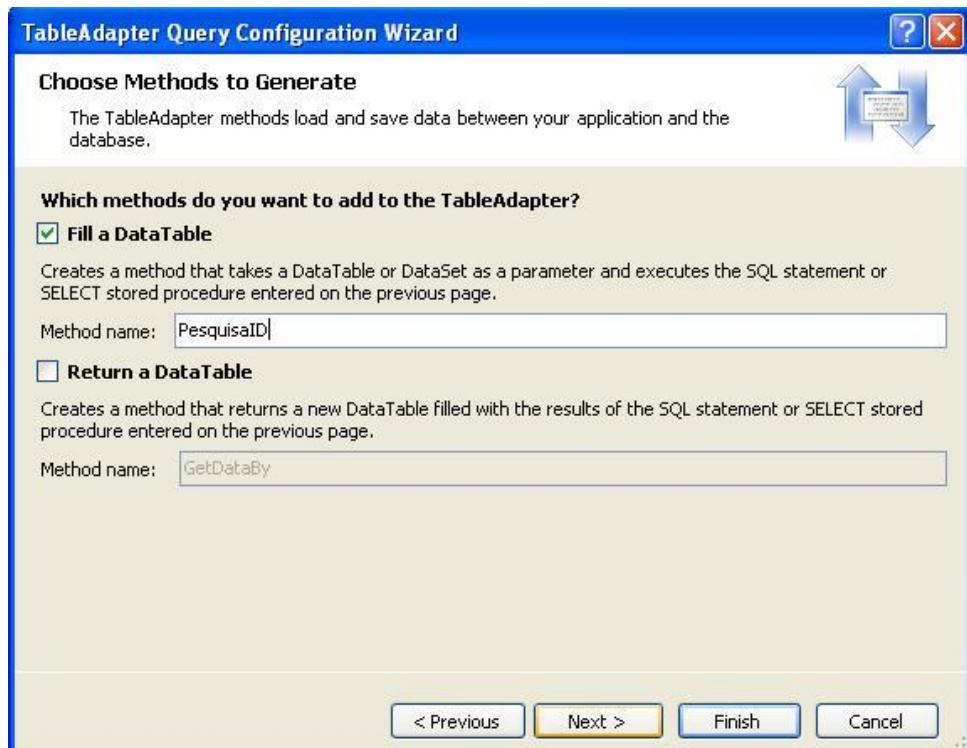
Na próxima tela, devemos informar qual tipo de consulta iremos realizar dentre cinco opções: se será por meio de um **SELECT** que retornará vários registros, se será por um **SELECT** que retornará apenas 1 registro, se iremos usar a instrução **UPDATE**, que atualiza os dados, se iremos usar a instrução **DELETE**, que apaga os dados ou se será por meio da instrução **INSERT**, que insere os dados. Escolha a 1ª opção e siga em frente.



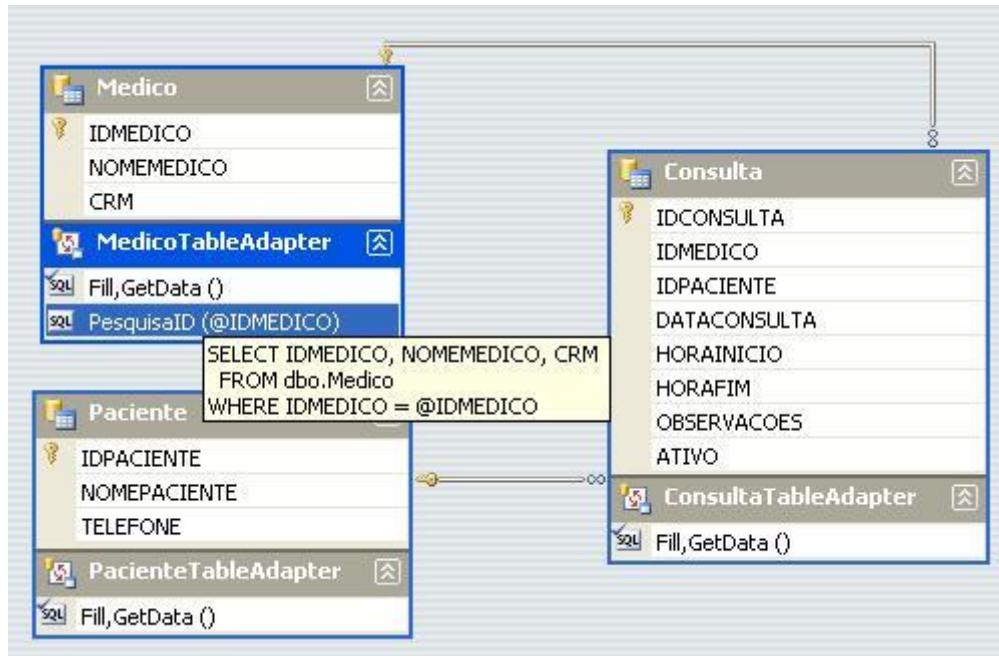
Aqui ele já me traz a instrução SQL pronta. Note o botão **Query Builder**, com ele podemos fazer instruções mais complexas, com o uso de **Join**, **Group By**, etc. Como a nossa consulta será parametrizada, volte a instrução SQL e altere a instrução passando o parâmetro como mostra a imagem a seguir:



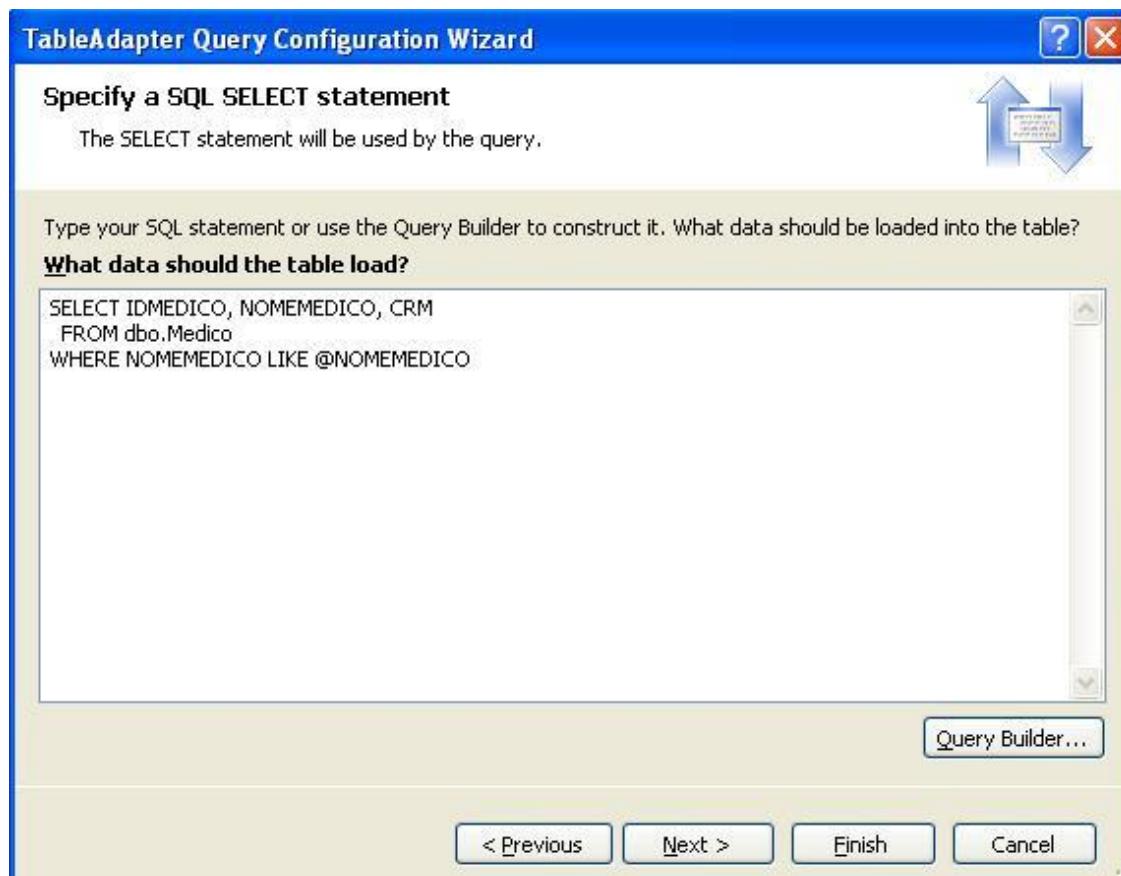
Nesta tela, devemos escolher quais métodos devemos usar em nosso TableAdapter. A 1^a opção, **Fill a DataTable**, como o próprio nome diz, preenche o DataTable e a 2^a opção, **Return a DataTable**, apenas retorna o DataTable preenchido. Podemos usar os dois, mas neste exemplo (como estou seguindo à risca as videoaulas de Luciano Pimenta) iremos escolher somente a 1^a opção e dar o nome de **PesquisaID**.



A próxima tela mostra que o Wizard gerou a instrução SQL e o método de preenchimento. Clique em **Finish**. Perceba na imagem abaixo que foi criada nossa query parametrizada, ou seja, ela pede como parâmetro o **ID do Médico**:



Agora crie um Wizard igual esse que criamos, apenas mude nossa **instrução SQL**, já que nesse pesquisaremos pelo nome. Como podemos pesquisar por apenas uma parte do nome, use a cláusula **Like**, como a imagem abaixo nos mostra:



Na próxima tela, selecione o método **Fill** e dê o nome **PesquisaNome**. Pronto, agora temos as duas consultas criadas. Agora vamos fazer um teste.

Primeiramente, dê um **Build Solution** (F6) na sua aplicação. Agora abra o **Form1.cs** e adicione o seguinte código:

```

9  using Consultas.dsConsultaTableAdapters;
10 
11 namespace Consultas
12 {
13     public partial class frmPrincipal : Form
14     {
15         public frmPrincipal()
16         {
17             InitializeComponent();
18         }
19 
20         private void frmPrincipal_Load(object sender, EventArgs e)
21         {
22             dsConsulta.MedicoDataTable dt = new dsConsulta.MedicoDataTable();
23             MedicoTableAdapter ta = new MedicoTableAdapter();
24 
25             ta.PesquisaID(dt, 1);
26             dt.Rows[0]["IDMEDICO"].ToString();
27         }
28     }

```

Fiz a chamada a meu DataSet por meio do **using** e, no método **Load** do meu formulário, instanciei o **MedicoDataTable**, instanciei o **MedicoTableAdapter**, que foi na onde criei minhas duas pesquisas e, por meio dele, fiz a chamada ao método **PesquisaID**, referente à pesquisa criada passando como parâmetro meu **MedicoDataTable** e um valor inteiro qualquer, nesse caso o 1. Por meio do meu método **Rows**, tenho acesso a coluna da minha tabela **Medico**, para isso usei o **dt**, que é a instância de meu **DataTable** e chamei o método **Rows**, passando como parâmetro o índice 0, que seria o 1º registro e nome da coluna, que neste caso é **ID_MEDICO**. Iremos fazer basicamente isso em nossas outras consultas. Esse exemplo iremos usar em nosso **Cadastro**, já a outra consulta que criamos usaremos em nosso formulário de **Pesquisa**.

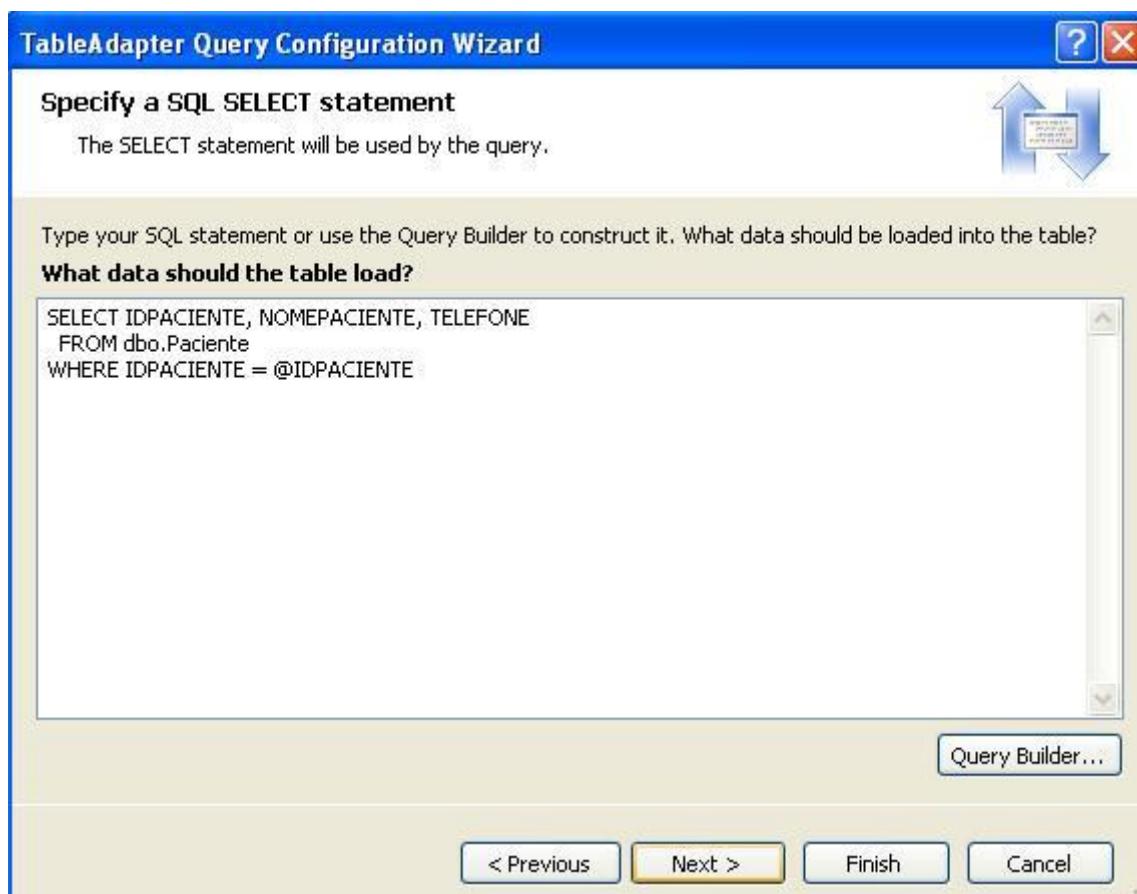
Finalizando, apague estes códigos do método **Load** desse form, já que isso nos serviu apenas de exemplo para implementarmos nas consultas e pesquisas dos outros formulários.

Na próxima parte de nossa série de artigos, iremos continuar a customização e implementação de métodos em nosso formulário de **Pesquisa** e iremos começar a criação do formulário base de **Cadastro**. Não perca!

Parte 7

Nesta parte iremos continuar a customização de nossas tabelas e sobrescrever os métodos para o formulário de **Cadastro de Pacientes**. Acompanhe.

Como feito no artigo anterior, vamos criar um **TableAdapter**, só que desta vez para a tabela de **Pacientes**. Então, clique com o botão direito no **TableAdapter** da tabela **Paciente** e clique em **Add Query**. Faça isso com os dois tipos de pesquisa, como no form de **Medicos**, por **ID** e por **Paciente**. Altere a instrução SQL da consulta por **ID**, como mostra a imagem abaixo e clique em **Next**:



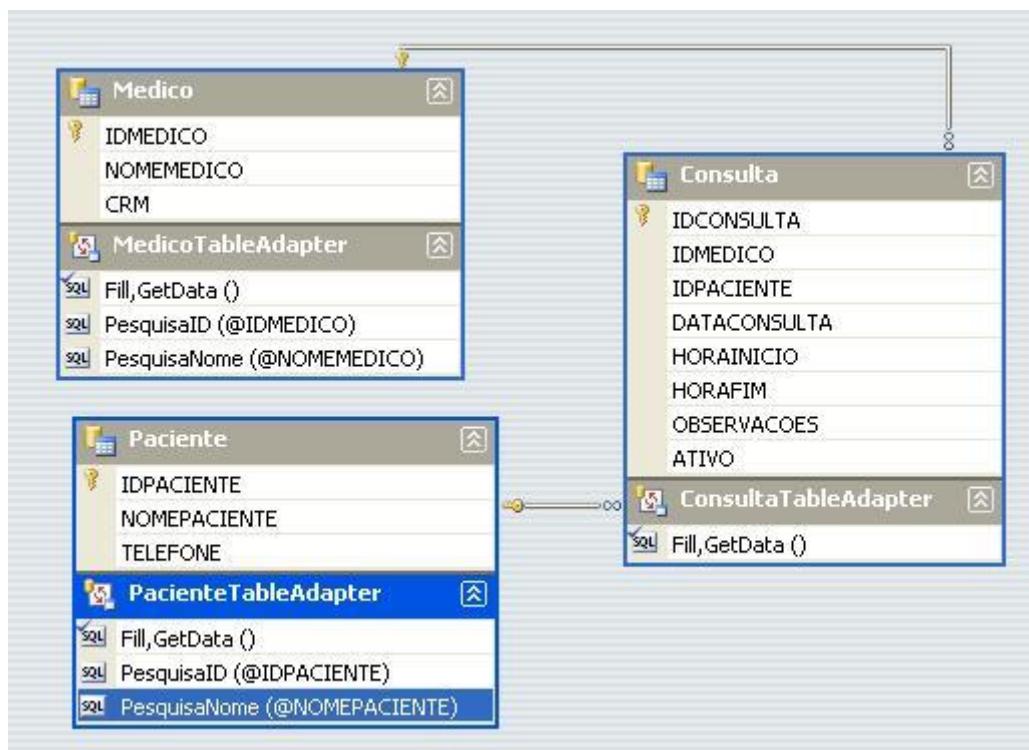
Na próxima tela, ao invés de escolhermos a opção **Fill a DataTable**, vamos escolher a outra opção, **Return a DataTable**, com o nome **PesquisaID** para vermos as diferenças. Escolha e clique em Next, depois em Finish.



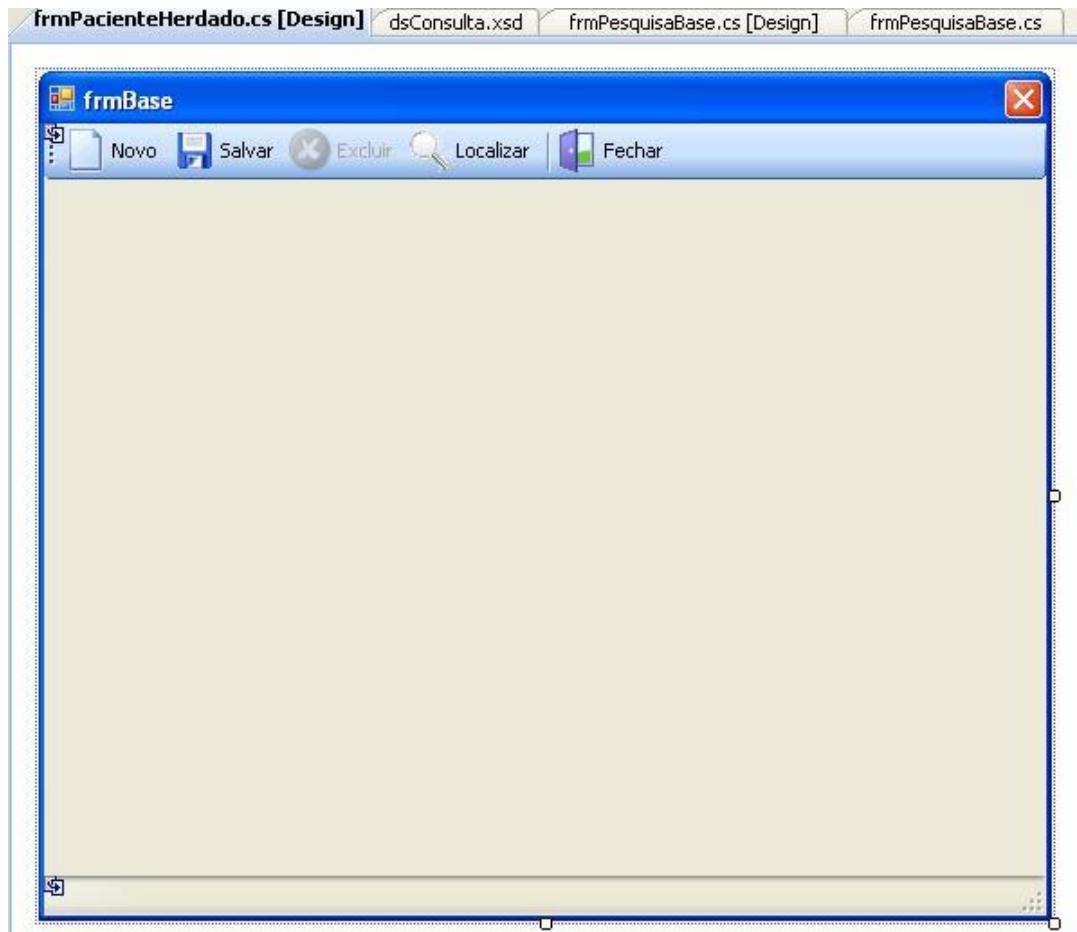
Agora crie outra query e faça o mesmo que a query anterior, só altere a instrução SQL, como a da imagem a seguir:



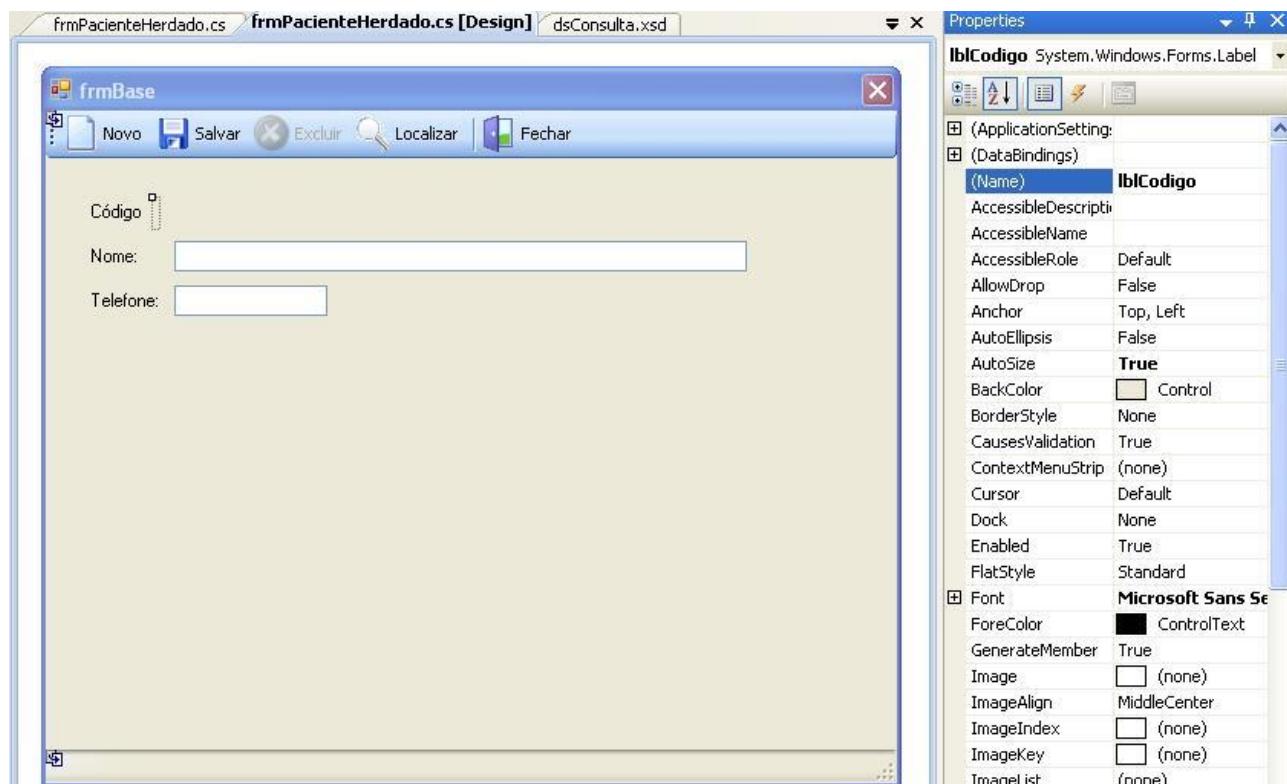
Escolha o método **Return a DataTable** e dê o nome de **PesquisaNome**. Pronto, nosso DataSet está concluído.



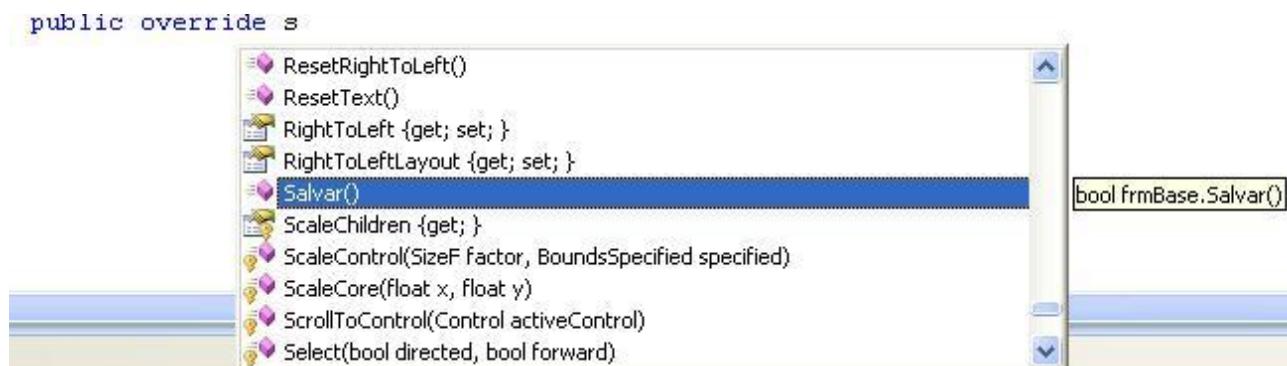
Agora vamos criar os formulários que herdarão dos formulários base que foram criados. Salve e dê um **Build Solution** (F6) no seu projeto para ter certeza de que o mesmo não está com erros. Agora clique com o botão direito no seu projeto e clique em **Add > New Item**. Não iremos adicionar um Windows Form normal e sim um **Inherited Form**, que traduzindo seria um Formulário Herdado que, como o próprio nome diz, herdará de outro formulário, neste caso do Paciente. Dê um nome sugestivo a ele e que não seja o mesmo nome dos formulário já criados. No exemplo dei o nome **frmPacienteHerdado**. Irá aparecer outra janela perguntando de qual form iremos herdar, escolha o **frmBase**, que é o que tem todas as características que precisamos, e clique em OK. Como você pode notar abaixo, temos os mesmos controles do frmBase, o que precisamos fazer no momento é adicionar alguns controles.



Adicione 4 labels e 2 textboxes, deixe o label que ficará em cima do textbox sem nome, pois ele será preenchido com os dados que vierem do banco. No modo design dê o nome **lblCodigo** à ele, **txtNome** e **txtTelefone** aos textboxes. Deixe seu formulário como o da imagem abaixo:



Agora precisamos sobrescrever os métodos que, no formulário base, foram declarados como virtual, como o **Salvar**, **Excluir**, **Localizar** e **Carregar Valores**. Para isso, aperte **F7** no **frmBase** para ir a página de códigos. O Visual Studio nos dá uma grande ajuda quando queremos sobreescrivemos métodos já criados. Quando digitamos **public override** e damos um espaço, o VS já nos traz todos os métodos que podemos utilizar do outro form, incluindo os que criamos, como você pode perceber na imagem a seguir:



Chame o método **Salvar** e exclua o **return base.Salvar()**, pois não iremos usá-lo. Aqui iremos chamar nosso DataSet criado anteriormente, antes disso declare-o por meio do **using**. Como meu método espera um valor booleano, eu preciso declarar uma variável do tipo **bool** e deixá-la como **false**, valor padrão. Após isso, instancio o **PacienteTableAdapter** e agora preciso chamar meu enumerador **sStatus**, se lembra dele? Mais aqui tem um problema: como ele foi declarado como **private**, não consigo chamá-lo aqui, já que ele é privado a sua classe. Para isso, altere o modificador dele para **public**. Agora sim, complete o método abaixo, veja a imagem com os comentários do que foi feito:

```

8 //sempre instanciar quando usar o TableAdapter
9 using Consultas.dsConsultaTableAdapters;
10
11 namespace Consultas
12 {
13     public partial class frmPacienteHerdado : Consultas.frmBase
14     {
15         public frmPacienteHerdado()
16         {
17             InitializeComponent();
18         }
19
20         private void frmPacienteHerdado_Load(object sender, EventArgs e)
21         {
22
23         }
24
25         public override bool Salvar()
26         {
27             bool bSalvar = false;
28             dsConsultaTableAdapters.PacienteTableAdapter ta = new PacienteTableAdapter();
29             //faço uma verificação, se o Status for Inserindo chamo o método Insert do
30             //TableAdapter e passo como parâmetro o txtNome e o txtTelefone. A variável
31             //bSalvar recebe este método Insert, desde que seja inserido ao menos 1 registro
32             if (sStatus == StatusCadastro.scInserindo)
33             {
34                 bSalvar = (ta.Insert(txtNome.Text, txtTelefone.Text) > 0);
35             }
36             return bSalvar;
37         }

```

Esta verificação é para quando estiver inserindo. Mais, e quando o registro já foi inserido e quero apenas atualizá-lo? Para isso, faço como na imagem a seguir (ainda dentro do meu método Salvar):

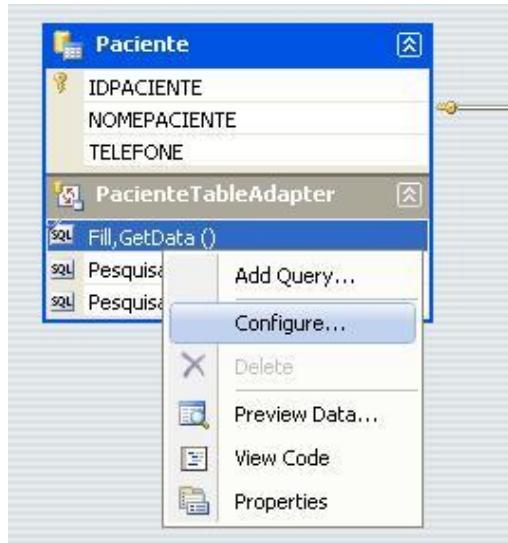
```

25         public override bool Salvar()
26         {
27             bool bSalvar = false;
28             dsConsultaTableAdapters.PacienteTableAdapter ta = new PacienteTableAdapter();
29             //faço uma verificação, se o Status for Inserindo chamo o método Insert do
30             //TableAdapter e passo como parâmetro o txtNome e o txtTelefone. A variável
31             //bSalvar recebe este método Insert, desde que seja inserido ao menos 1 registro
32             if (sStatus == StatusCadastro.scInserindo)
33             {
34                 bSalvar = (ta.Insert(txtNome.Text, txtTelefone.Text) > 0);
35             }
36             //se o Status for Navegando, tenho que atualizar o registro, para isso faço igual ao
37             //método Insert, só que aqui meu
38             else if (sStatus == StatusCadastro.scEditando)
39             {
40                 bSalvar = (ta.Update(txtNome.Text, txtTelefone.Text, int.Parse(lblCodigo.Text)) > 0);
41             }
42             string Label.Text
43             return bSalvar;
44         }

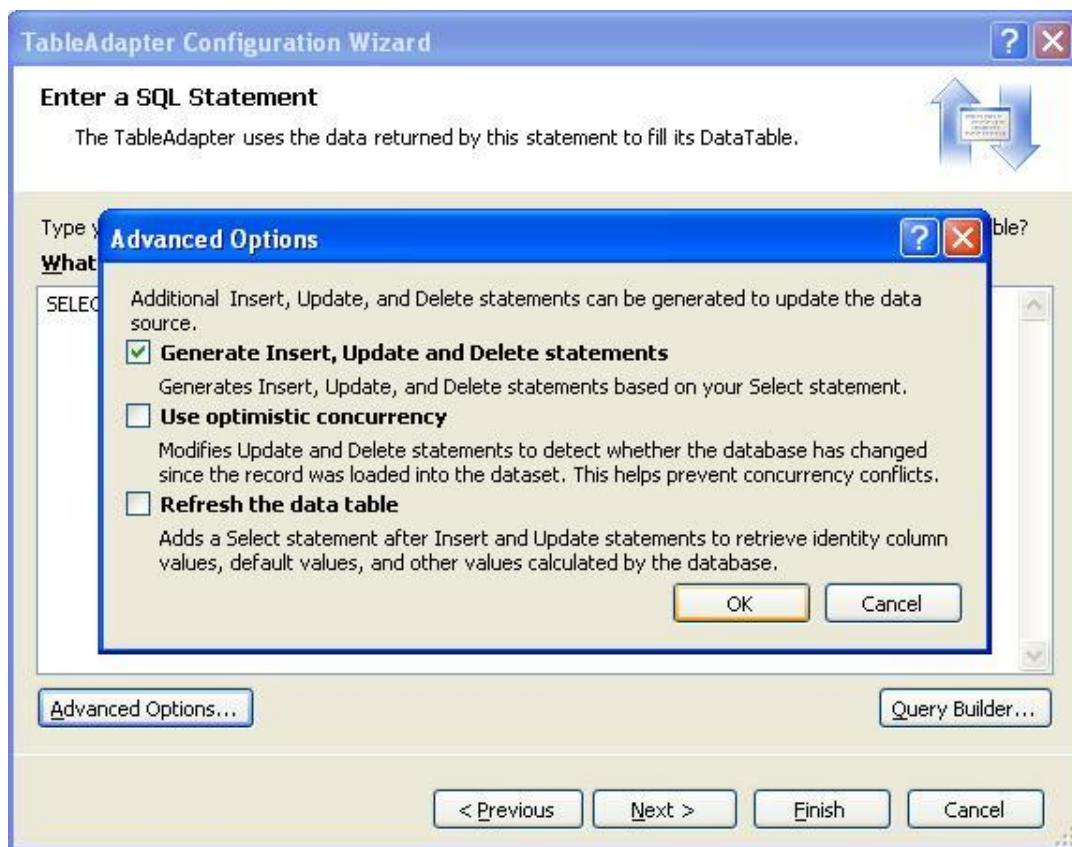
```

Error:
No overload for method 'Update' takes '3' arguments

Como você pode perceber, o método **Update** vai dar erro porque ele espera ao menos 5 parâmetros a serem preenchidos. Porque ele está pegando todos os campos de minha tabela e está tentando comparar com os campos do meu método. Para alterarmos isso, temos que ir ao DataSet, clicarmos com o botão direito em **Fill, GetData()**, de meu PacienteTableAdapter, e clicarmos em **Configure**, como mostra a imagem:



Agora clique em Advanced Options e desmarque as opções **Use optimistic concurrency** e **Refresh the data table**, como você pode ver abaixo:



De OK, Next, Next e Finish. Volte ao método **Update** (dentro do método **Salvar**) e note que agora não irá mais dar erro. Veja como deve ficar método completo abaixo.

```

25     public override bool Salvar()
26     {
27         bool bSalvar = false;
28         dsConsultaTableAdapters.PacienteTableAdapter ta = new PacienteTableAdapter();
29         //faz uma verificação, se o Status for Inserindo chamo o método Insert do
30         //TableAdapter e passo como parâmetro o txtNome e o txtTelefone. A variável
31         //bSalvar recebe este método Insert, desde que seja inserido ao menos 1 registro
32         if (sStatus == StatusCadastro.scInserindo)
33         {
34             bSalvar = (ta.Insert(txtNome.Text, txtTelefone.Text) > 0);
35         }
36         //se o Status for Navegando, tenho que atualizar o registro,
37         //para isso faço igual ao método Insert, só que aqui passo
38         //também minha label com o código que virá com os dados do banco
39         else if (sStatus == StatusCadastro.scEditando)
40         {
41             bSalvar = (ta.Update(txtNome.Text, txtTelefone.Text, int.Parse(lblCodigo.Text)) > 0);
42         }
43
44         return bSalvar;
45     }

```

Dê um **Build Solution** (F6) apenas para ter certeza que sua solução está sem erros e salve o projeto.

Agora vamos sobreescrivar o método Excluir. Faça como o anterior:

```

public override bool Excluir()
{
    bool bExcluir = false;
    dsConsultaTableAdapters.PacienteTableAdapter ta = new PacienteTableAdapter();

    bExcluir = (ta.Delete(int.Parse(lblCodigo.Text)) > 0);

    return bExcluir;
}

```

Apenas chame o método Localizar, já que não vamos implementá-lo agora.

Agora, antes de chamar o método que carrega os itens vamos imaginar o seguinte: já que em meus métodos eu preciso passar meu **lblCodigo**, referente ao código do paciente, não seria melhor se eu declarasse uma variável que fosse a responsável por este código? Pensando assim, vamos criá-la no nosso formulário base. Abra o **frmBase**, vá a página de código e declare o código:

```
public int _nCodGenerico;
```

Desta forma, todos os meus cadastros herdarão essa variável, que representa o código respectivo de cada cliente de meu consultório. Assim, posso usá-la nos métodos **Localizar**, **Excluir** e **Salvar**, substituindo pelo **lblCodigo**, já que não precisamos usá-lo mais. Por enquanto vamos deixar como está.

Agora chame o método **CarregaValores** e dentro dele coloque o seguinte código:

```

public override void CarregaValores()
{
    //instancio o DataTable e o TableAdapter, meu DataTable recebe o TableAdapter
    //que usa o método PesquisaID, passando como parâmetro a variável nCodGenerico
    dsConsulta.PacienteDataTable dt = new dsConsulta.PacienteDataTable();
    PacienteTableAdapter ta = new PacienteTableAdapter();
    dt = ta.PesquisaID(_nCodGenerico);

    //verifico se retornou algum registro. Se sim, preencho os controles de tela
    if (dt.Rows.Count > 0)
    {
        lblCodigo.Text = dt.Rows[0]["IDPACIENTE"].ToString();
        txtNome.Text = dt.Rows[0]["NOMEpaciente"].ToString();
        txtTelefone.Text = dt.Rows[0]["TELEFONE"].ToString();
    }
}

```

Aqui usamos aquela consulta que filtra pelo ID do Paciente, que criamos anteriormente em nosso DataSet. Para isso, declarei o **DataTable** e o **TableAdapter** e fiz com que o DataTable retornasse meu TableAdapter com o método **PesquisaID** passando como parâmetro minha variável **nCodGenerico**, que será preenchida pelo método **Localizar**, lembrando que o Localizar que irá chamar meu método **CarregaValores**.

Após isso eu verifico, se o DataTable retornou algum registro, preencho os controles de tela. Lembrando que esse CarregaValores será implementado em cada formulário específico.

Vamos criar agora um novo formulário, que herdará do de Pesquisa. Vá na Solution, clique em **Add > New Item**, em **Categories** escolha Windows Forms, escolha o template **Inherited Form**, dê o nome de **PesquisaPaciente**, clique em Add e escolha o **frmPesquisaBase**, já que nosso form criado herdará dele.

Volte ao formulário que estávamos, porque agora vamos implementar o método **Localizar**. Digite o seguinte código:

```

public override bool Localizar()
{
    bool bLocalizar = false;
    frmPesquisaPaciente frmPesquisa = new frmPesquisaPaciente();
    //faço abaixo uma verificação, se o usuário clicar em OK
    //minha variável bLocalizar recebe o sCdCodigo
    if (frmPesquisa.ShowDialog() == DialogResult.OK)
    {
        bLocalizar = (frmPesquisa.sCdCodigo != "");
        //verifico agora se meu bLocalizar retornou algum registro
        if (bLocalizar)
        {
            _nCodGenerico = int.Parse(frmPesquisa.sCdCodigo);
        }
    }

    return bLocalizar;
}

```

Instanciei o formulário herdado de Paciente, verifiquei se o **DialogResult** desse form foi igual a OK como configurado no formulário base, ou seja, se o usuário escolheu algum registro no formulário e clicou em OK e, se sim, passamos a variável **bLocalizar** o valor daquela variável **sCdCodigo** (que é a responsável pelo código da pesquisa no formulário base), diferente de vazia. Após isso, fazemos outra verificação. Se isso for verdade, ou seja, se a variável **bLocalizar** obtiver algum registro, minha variável **nCodGenerico** receberá o valor

de **sCdCodigo**. Acima fiz também um **int.Parse** para converter, pois a variável **sCdCodigo** é do tipo **string** e a **nCodGenerico**, do tipo **int**.

Exemplo: o usuário está navegando entre os registros, decide escolher um e clica em OK. Neste momento meu **DialogResult** é “setado” para OK. Quando isso acontece, faço estas verificações, o **bLocalizar** recebe o valor de **sCdCodigo**, que na verdade é o código escolhido. Ele é diferente de vazio? é, então foi preenchido, o que significa que minha outra variável, a **nCodGenerico** irá receber o **sCdCodigo**. Simples né?!

A partir de agora, podemos utilizar esta variável **nCodGenerico** em outros locais para substituir meu código. No método **CarregaValores** já estamos usando ela. Vamos fazer o mesmo no método **Excluir** e no método **Salvar**:

```
public override bool Excluir()
{
    bool bExcluir = false;
    dsConsultaTableAdapters.PacienteTableAdapter ta = new PacienteTableAdapter();

    bExcluir = (ta.Delete(_nCodGenerico) > 0);

    return bExcluir;
}

public override bool Salvar()
{
    bool bSalvar = false;
    dsConsultaTableAdapters.PacienteTableAdapter ta = new PacienteTableAdapter();
    //faço uma verificação, se o Status for Inserindo chamo o método Insert do
    //TableAdapter e passo como parâmetro o txtNome e o txtTelefone. A variável
    //bSalvar recebe este método Insert, desde que seja inserido ao menos 1 registro
    if (sStatus == StatusCadastro.scInserindo)
    {
        bSalvar = (ta.Insert(txtNome.Text, txtTelefone.Text) > 0);
    }
    //se o Status for Navegando, tenho que atualizar o registro,
    //para isso faço igual ao método Insert, só que aqui passo
    //também minha label com o código que virá com os dados do banco
    else if (sStatus == StatusCadastro.scEditando)
    {
        bSalvar = (ta.Update(txtNome.Text, txtTelefone.Text, _nCodGenerico) > 0);
    }

    return bSalvar;
}
```

Nosso formulário de Cadastro está implementado.

Finalizando nosso artigo, vá ao **Form1**, dê dois cliques no botão Paciente e altere o código para chamar nosso formulário. Agora será chamado nosso formulário herdado. Faça como a imagem abaixo nos mostra:

```
private void toolStripButton2_Click(object sender, EventArgs e)
{
    frmPacienteHerdado frmPaciente = new frmPacienteHerdado();
    frmPaciente.ShowDialog();
}
```

Salve seu projeto e compile para testar se tudo saiu OK.

Na próxima parte de nossa série de artigos, iremos criar o método Pesquisar, do formulário de Pesquisa, para que assim nosso Cadastro de Paciente esteja completo. Tudo isso usando herança

visual. Iremos também começar a fazer uma simulação do padrão **MVC (Model View Controller)** em nosso sistema. Não perca!

Parte 8

Nesta parte iremos implementar o método Pesquisar, do formulário de Pesquisa, referente ao **Cadastro de Pacientes** para que, assim, nosso cadastro esteja completo usando os conceitos de Herança Visual de formulários. Iremos começar também uma simulação do padrão **MVC (Model View Controller)** em nosso sistema. Confira:

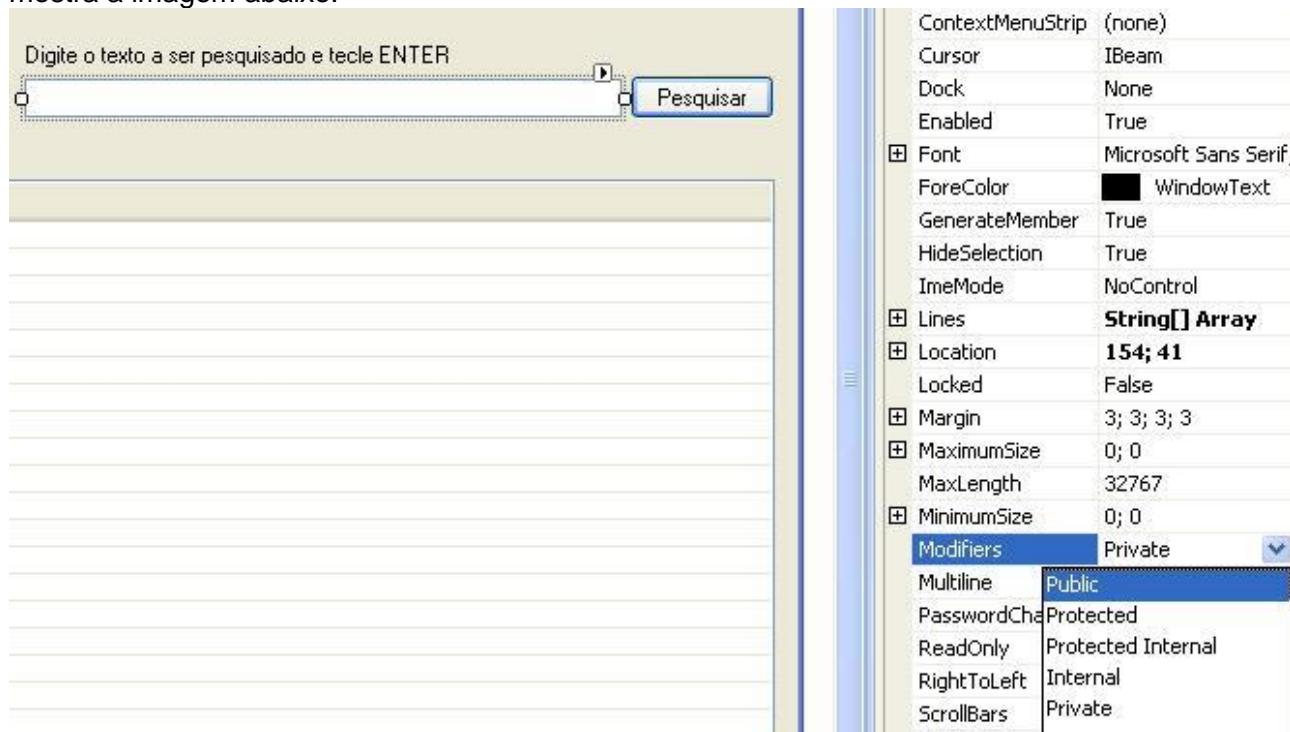
No formulário de Pesquisa, vamos fazer o seguinte: de acordo com o tipo de pesquisa que o usuário escolher (por ID ou Descrição), iremos utilizar um dos métodos criados anteriormente em nosso DataSet.

Abra o formulário **frmPesquisaPaciente** e sobrescreva o método Pesquisar, como mostra a seguir:

```
public override void Pesquisar()
{
    base.Pesquisar();
}
```

Após isso, dentro do método criado, tente acessar o campo de pesquisa, herdado do formulário base, que se chama **txtPesquisa**. Você verá na imagem Ao lado que não será possível o acesso ao mesmo.

Isso acontece porque, por padrão, o modificador de acesso dos controles é do tipo privado (**private**), precisamos então acessar o formulário base e alterá-lo para público (**public**), como mostra a imagem abaixo:



Faça o mesmo com os RadioButtons **Código** e **Descrição** e aperte **F6** para dar um **Build Solution**. Toda vez que você precisar acessar um controle herdado desta forma, é preciso alterar o modificador para **public** no formulário base.

Agora volte ao formulário de pesquisa e note que você terá acesso ao textbox de pesquisa e aos radiobuttons. Altere o nome de seu formulário de pesquisa para **Pesquisar Paciente** e volte ao código.

Dentro do método sobreescrito **Pesquisar**, faça o seguinte código:

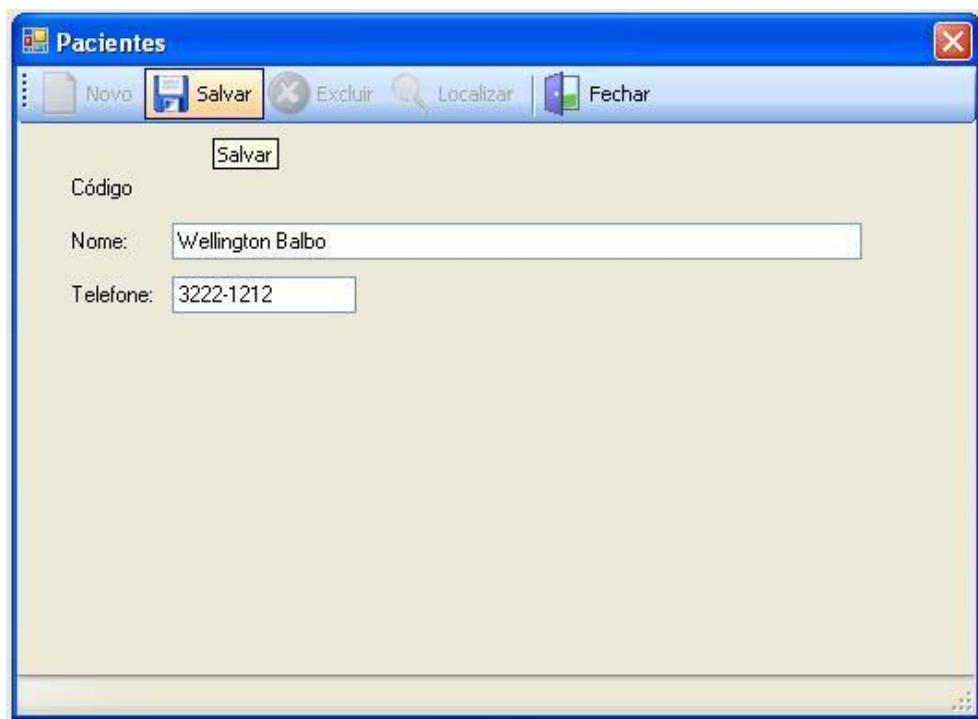
```
public override void Pesquisar()
{
    //instancio o DataTable e o TableAdapter
    dsConsulta.PacienteDataTable dt = new dsConsulta.PacienteDataTable();
    dsConsultaTableAdapters.PacienteTableAdapter ta =
        new Consultas.dsConsultaTableAdapters.PacienteTableAdapter();

    //verifico se foi selecionado o radiobutton código ou descrição e
    //passo os parâmetros necessários para que seja feita a pesquisa
    if (rbtCodigo.Checked)
    {
        dt = ta.PesquisaID(int.Parse(txtPesquisa.Text));
    }
    else
    {
        dt = ta.PesquisaNome("%" + txtPesquisa.Text + "%");
    }

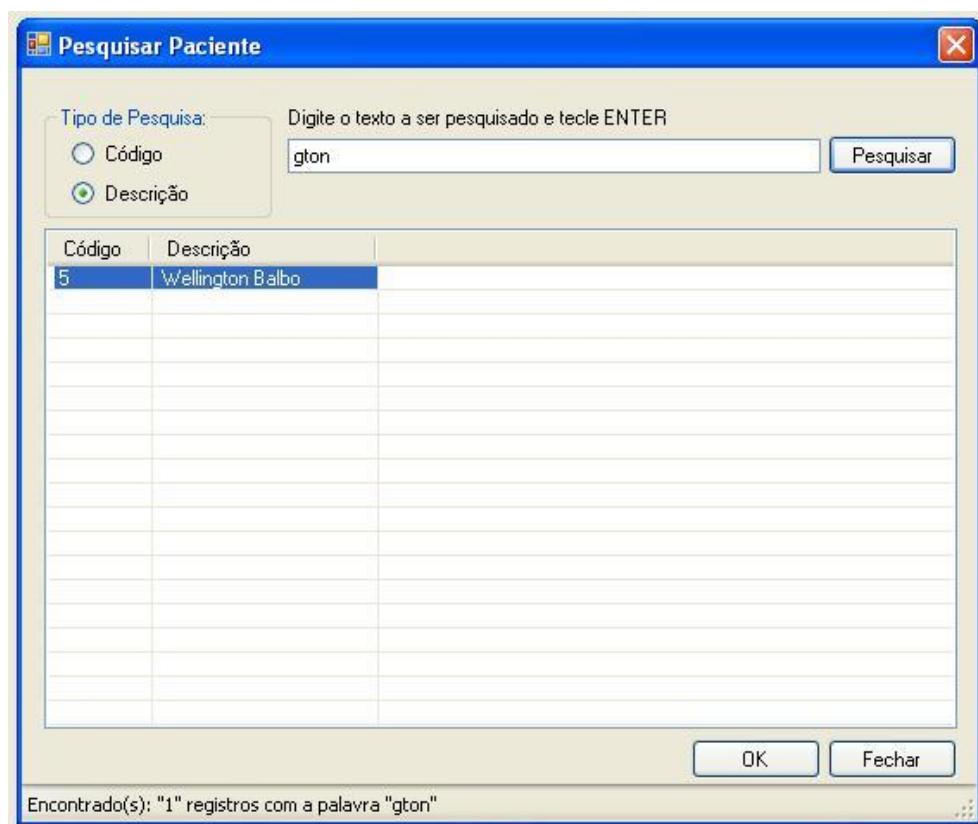
    //após feita a verificação chamo o método CarregarItens
    //passando como parâmetro o DataTable instanciado
    CarregarItens(dt);
}
```

O que fiz acima foi uma verificação simples, Se o usuário selecionar o RadioButton **Código**, passo ao DataTable o método **PesquisaID**, criado no DataSet anteriormente, para ser pesquisado o ID que o usuário digitar no campo de pesquisa. Senão, ou seja, se o usuário selecionar o RadioButton **Descrição** passo ao DataTable o método **PesquisaNome**, e uso a porcentagem (%) entre o campo de pesquisa, que significa que ele irá procurar uma parte do nome.

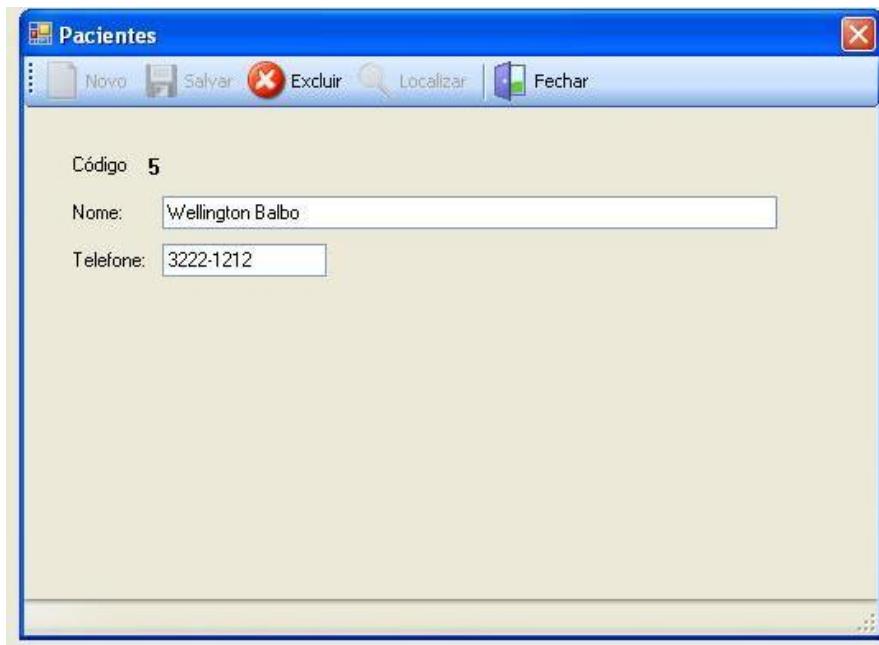
Se eu tiver no banco um nome Carlos, por exemplo e digitar no campo **rlo**, ele irá me retornar o nome Carlos no ListView, pois com esse parâmetro de porcentagem, da cláusula **Like** (que irei abordar mais a frente no [curso de SQL](#) do meu blog), ele irá procurar por partes referentes ao nome pesquisado. Como meu DataSet não aceita/entende o parâmetro %, preciso declará-lo em meu código. Agora salve e compile sua aplicação. Clique em Pacientes e deverá aparecer a tela de Cadastro de Paciente. Clique em **Novo**, digite um nome e um telefone qualquer e clique em **Salvar**.



Agora clique no botão **Localizar** para ir ao formulário de Pesquisa:



Acima digitei uma parte do nome, ele já me retornou o nome completo e o ID. E abaixo me mostrou quantos registros foram retornados com essa parte do nome. Se você der dois cliques em cima do nome, ele trará a tela de **Cadastro** com o Nome e o ID preenchido:

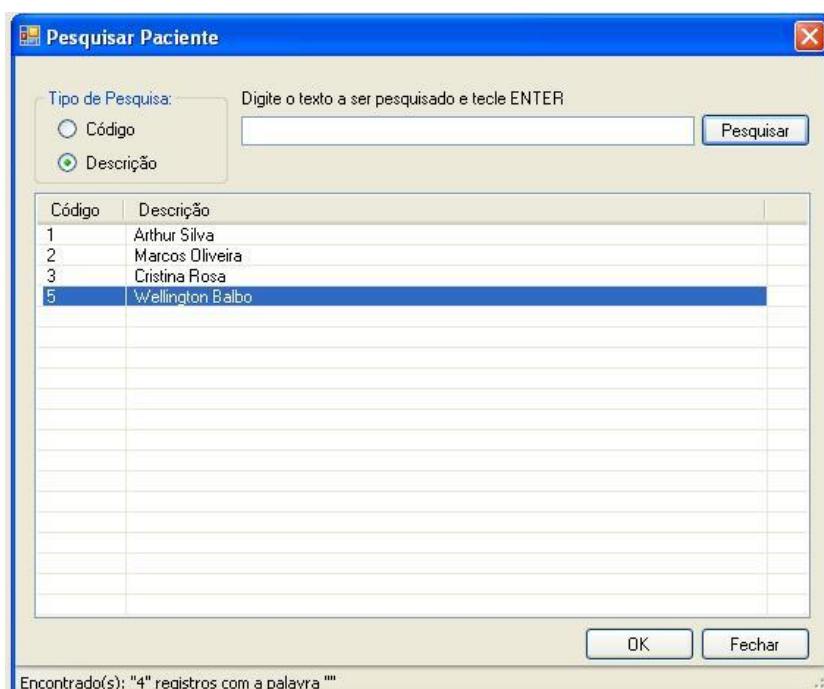


Como você percebe na imagem acima, os botões de Salvar e de Localizar estão desabilitados, precisamos alterar algumas configurações para que os botões sejam habilitados. Para isso, vá ao formulário **Base**, vá ao método **HabilitaDesabilitaControles** e perceba que no botão **Localizar** só está habilitado o Status **Navegando**, adicione também o **Editando** e o **Inserindo**. No botão Salvar, está habilitado o Status **Navegando** e **Inserindo**, falha minha, o correto seriam os Status **Editando** e **Inserindo**. Arrume os botões como mostra a imagem a seguir:

```
//Localizar
btnLocalizar.Enabled = (sStatus == StatusCadastro.scNavegando ||
                        sStatus == StatusCadastro.scEditando ||
                        sStatus == StatusCadastro.scInserindo);

//Salvar
btnSalvar.Enabled = (sStatus == StatusCadastro.scEditando ||
                      sStatus == StatusCadastro.scInserindo);
```

Salve e compile seu projeto. Clique em **Pacientes** e no botão **Localizar**. Digite um nome qualquer na caixa de busca ou a deixe em branco e tecle **ENTER**. Você verá o resultado da pesquisa:



Clique em um dos resultados da pesquisa. Se quisermos com que, quando o usuário teclar **ESPAÇO** sobre um dos resultados, o formulário de pesquisa se feche e abra o respectivo registro, podemos implementar isso fazendo o seguinte: Abra o formulário de **Pesquisa Base**, clique em cima do **ListView**, vá a janela de **Propriedades** e, no evento **KeyDown**, dê dois cliques e adicione o seguinte código:

```
private void lstPesquisa_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Space)
    {
        sCdCodigo = lstPesquisa.SelectedItems[0].Text;
        Close();
        DialogResult = DialogResult.OK;
    }
}
```

Nota:

Tentei fazer este método com a tecla ENTER, mas por uma razão desconhecida, não consegui fazer funcionar. Se alguém descobrir o porque, por favor poste nos comentários.

Agora escolha um registro qualquer, abra-o e clique em Excluir. Como você pode perceber, a exclusão é feita em poucos segundos. O interessante seria perguntarmos ao usuário se ele realmente deseja excluir o registro, mantendo assim uma certa padronização em formulários. Para isso, vá ao formulário **Base**, entre no botão **Excluir**, e altere o código para que fique desta forma:

```
private void btnExcluir_Click(object sender, EventArgs e)
{
    if (MessageBox.Show("Deseja excluir o registro?", "Excluir",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.No)
    {
        return;
    }
    else
    {
        if (Excluir())
        {
            sStatus = StatusCadastro.scNavegando;
            LimpaControles();
            HabilitaDesabilitaControles(false);
            MessageBox.Show("Registro excluído com sucesso", "Aviso do Sistema",
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
        else
        {
            MessageBox.Show("O registro não foi excluído, por favor verifique os erros!", "Erro",
                MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
}
```

Acima fiz apenas a pergunta ao usuário se deseja excluir o registro com o uso de **MessageBox**. Se ele clicar em não, nada acontecerá. Se ele disser sim, ele entrará no método **Excluir**, que já foi implementado anteriormente e o usuário verá um **MessageBox** avisando que o registro foi excluído. Simples.

No decorrer dos artigos iremos verificar o que está faltando ou até os erros que possam vir a ocorrer para que assim possamos alterá-los/modificá-los e nosso sistema fique com o menor número de inconsistências possíveis.

Salve o projeto, compile, clique para excluir um registro e veja as modificações.

Vamos começar a partir de agora uma simulação do padrão **MVC (Model View Controller)**, onde nosso **DataSet** será o **Model** e os formulários são o nosso **View**. Precisamos criar agora o **Controller**,

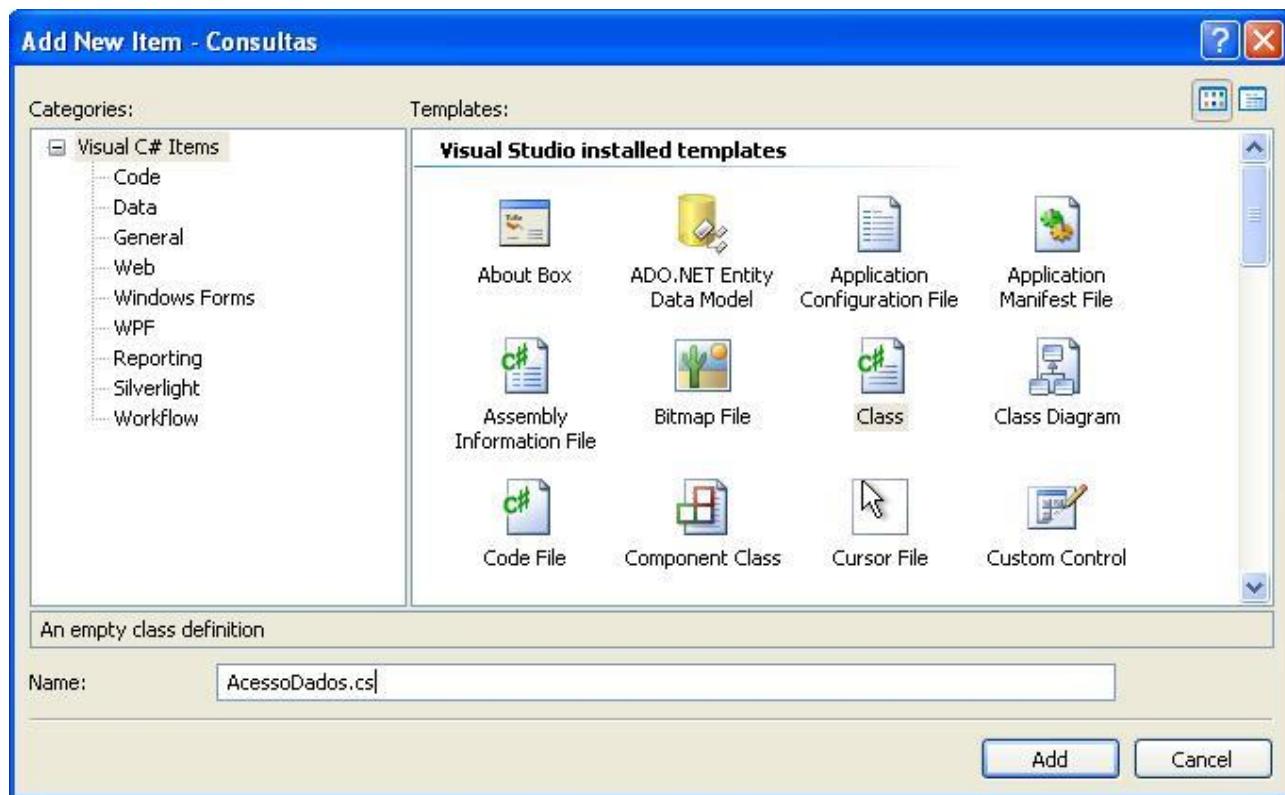
que serão as classes que farão a abstração dos dados da Model e irá fazer a “ponte” com a View, ou seja, com essas classes poderemos abstrair os dados do nosso DataSet e onde faremos a implementação dos métodos que os formulários irão chamar.

Para mais informações sobre o padrão MVC, acesse os links abaixo:

- [MVC](#)
- [O modelo MVC - Model View Controller](#)
- [ASP.NET MVC](#)
- [Considerações iniciais sobre o ASP.NET MVC Framework](#)

Começaremos criando a classe de acesso a dados, que irá implementar alguns métodos e criaremos as classes que herdarão estes métodos.

Na **Solution Explorer**, clique em **Add > Class** e dê o nome **AcessoDados** a sua classe.



Lembrando o conceito de Herança, esta classe será nossa classe Base, onde a partir dela serão criados outras classes que herdarão os métodos provenientes dela. Aqui iremos ter os métodos para **Salvar**, para **Excluir**, **Pesquisa por ID** e **Pesquisa por Nome**, seguindo nossa lógica do sistema.

No nome da sua classe acrescente um **public** antes, para que ela possa ser acessada pelas classes herdadas e crie o método virtual de Salvar, que será sobrescrito depois.

```
public class AcessoDados
{
    public virtual bool Salvar(bool bInsert)
    {
        return false;
    }
}
```

Como você pode perceber, o método **Salvar** recebe um parâmetro booleano, que irá indicar, dependendo do status, se o registro está sendo inserido ou alterado. Como fiz nos outros métodos do tipo **bool**, inicializarei este método com o valor **false** como retorno. Crie também um método virtual booleano com o nome **Delete**. Este método não terá nenhum parâmetro, já que a função é apenas de excluir um registro.

```
public virtual bool Delete()
{
    return false;
}
```

Agora vamos criar um método do tipo **DataTable** para pesquisar por ID, que receberá como parâmetro um valor do tipo **int**. Antes disso, é preciso declarar o namespace **System.Data**, para que possamos usar o tipo **DataTable**.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Data;
```

Após isso, faça o seguinte método:

```
public virtual DataTable PesquisaID (int nCodGenerico)
{
    return null;
}
```

Faça também um método do tipo **DataRow**, que irá me retornar somente uma linha, para pesquisar pelo ID, passando como parâmetro um valor do tipo **int**, parecido com o método anterior. A diferença deste é que irá me retornar um **DataRow** e não um **DataTable**.

```
public virtual DataRow PesquisaID ()
{
    return null;
}
```

Assim podemos fazer pesquisas que irão retornar somente uma linha e também pesquisas que irão retornar diversas linhas. Desta forma tenho várias formas de pesquisar em meu sistema. Desta forma, posso declarar dois métodos com o mesmo nome, só que com o retorno e parâmetro diferente (ou nesse caso, sem parâmetro).

Finalizando, faça um método do tipo **DataTable** para pesquisar por nome, que terá um parâmetro do tipo **string**, que receberá o nome pesquisado.

```
public virtual DataTable PesquisaNome (string sDsNome)
{
    return null;
}
```

Assim teremos métodos para Salvar, Excluir, Pesquisar por ID e Pesquisar por Nome.

Na próxima parte de nossa série de artigos, iremos criar as classes que herdarão dessa minha classe de acesso a dados e assim implementaremos as funcionalidades para que nossa **View** não tenha contato com o banco de dados, ou seja, nossos formulários acessarão os dados através do **Controller**, que são nossas classes. Não perca!

9. Conceitos de Bancos de Dados Não-Relacionais (NoSQL)