

Bancos de dados relacionais

Aprendendo praticando



Alexandre Rozante

Sumário

Prefácio	5
Introdução	6
Capítulo I – Conceitos iniciais, parte I.....	7
Banco de Dados.....	7
Sistema Gerenciador de Bancos de Dados - SGBD	11
SGDB relacionais mais populares	12
Linguagem SQL - surgimento	13
O que vimos nesse capítulo	14
Capítulo II - Conceitos iniciais, parte II	15
Esquema.....	15
Tabela	15
Registro.....	16
Campo	16
Tipo de dado	17
CHAR x NCHAR e VARCHAR e NVARCHAR	19
CHAR ou VARCHAR?	20
AUTOINCREMENT	20
Página.....	21
Índice	22
Chave Primária.....	24
Relacionamento	25
Chave Estrangeira	27
Alteração e Exclusão com chaves estrangeiras	28
Chave Única	29
O que vimos nesse capítulo	31
Capítulo III – Primeiros passos práticos.....	32
Instalação do SGBD MySQL Server.....	32
Instalação do utilitário MySQL Workbench	34
Criação do Banco de Dados	37
Definição do usuário de acesso	43
Criação de Schemas	45

Criação de usuários adicionais.....	49
O que vimos nesse capítulo	51
Capítulo IV – Criação de Tabelas	52
SQL e criação de tabelas.....	53
Definições de campos.....	54
Restrições	57
Primary key.....	57
Foreign key.....	57
Not NULL	57
Unique	57
Check	57
DROP TABLE.....	61
O que vimos nesse capítulo	62
Capítulo V – O modelo de dados a criar	63
Exemplo	67
Capítulo VI – Solução para o modelo proposto.....	70
Explicando.....	73
Tabelas “montadora” e “veículo”	73
Tabela “componente”	73
Tabela “fornecedor”	74
Tabela “estoque”	75
Tabela “vendedor”	77
Tabela “venda”	77
Tabela “item_venda”	78
Otimização das consultas.....	80
O que vimos nesse capítulo	81
Capítulo VII – Mais comandos DDL	82
CREATE (UNIQUE) INDEX	82
DROP INDEX.....	83
ALTER TABLE	83
ADD COLUMN	83
DROP COLUMN	83
ADD CONSTRAINT	84
DROP CONSTRAINT.....	84

O que vimos nesse capítulo	85
Capítulo VIII – Testando o esquema	86
Scripts SQL.....	86
DDL x DML.....	87
INSERT	89
UPDATE	93
DELETE	94
START TRANSACTION, ROLLBACK e COMMIT	95
Transação	95
Transações e campos autoincrementados	98
Transações e comandos DDL.....	98
Transações e concorrência	98
SELECT – Parte I.....	99
O que vimos nesse capítulo	101
Capítulo IX – Indo além das consultas simples	102
SELECT – parte II	102
Uso de funções e expressões	103
CASE WHEN.....	108
LIKE.....	108
SELECT ... GROUP BY	109
HAVING.....	112
JOIN.....	112
INNER JOIN.....	115
OUTER JOIN.....	117
RIGHT (OUTER) JOIN	117
Aliás de nome de tabela	118
O que vimos nesse capítulo?	120
Capítulo X – Mais sobre SELECT.....	121
Limit ou Top.....	121
Distinct.....	122
Subqueries.....	124
O que vimos nesse capítulo	126
Capítulo XI – VIEWS.....	127
Pontos de atenção	129

DESCRIBE	129
EXPLAIN	130
O que vimos nesse capítulo	131
Capítulo XII – Triggers.....	132
O que vimos nesse capítulo	141
Capítulo XIII – Stored Procedures	142
CREATE PROCEDURE	142
CALL	143
DROP PROCEDURE	143
O que vimos nesse capítulo	144
Capítulo XIV – Tópicos finais	145
Coalesce().....	146
Sequences e Generators.....	148
Colunas BLOB e CLOB	149
Particionamento	149
Backup e Restore	150
Clusters.....	152
O que vimos nesse capítulo	153
Capítulo XV – Conexão SGBD x Programas	154
Drivers.....	154
O que vimos nesse capítulo	156
Capítulo XVI – Conclusão	157

Prefácio

Ao longo da minha carreira não foram poucas as vezes que deparei com amigos que gostariam de entender melhor sobre bancos de dados. Reparei que muitos tentaram, mas sentiram-se intimidados e acabaram desistindo. Há também uma parcela de amigos e amigas, que acabaram desmotivados, por enfrentarem dificuldade demais em cursos com carga horária inadequada, explicações superficiais demais e outros fatores.

Então, como um desafio pessoal, cuja única pretensão é tentar ajudar esses amigos e amigas, resolvi elaborar esse pequeno guia. A ideia é tentar usar uma abordagem diferente, que seja mais fácil de compreender e gere motivação para seguir adiante, a cada etapa.

Talvez não cumpra esse objetivo aqui, e se esse for o caso, peço desculpas antecipadamente. Contudo, se conseguir ajudar ao menos uma pessoa, já valeu!

Introdução

Ao longo desse guia você encontrará alguns trechos especiais. Para identificá-los são empregadas algumas convenções, explicadas a seguir:



Esse símbolo sinaliza explicações adicionais e dicas que considero úteis.



Referências para materiais complementares, downloads e websites.



Ponto de atenção. Algo que pode ser perigoso ou que pode gerar alguma dificuldade adicional.

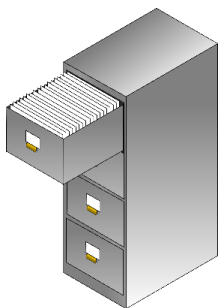
Adicionalmente, teremos comandos e trechos de programação, que serão identificados através de cor e fonte de caracteres diferentes, como nesse exemplo:

`comando a executar`

Capítulo I – Conceitos iniciais, parte I

Banco de Dados

Bancos de dados são coleções de dados estruturados, organizados para proporcionar performance e integridade. Podemos empregar uma analogia simples para começar a compreendê-los:



Sim! O bom e velho armário-de-aço para pastas suspensas.

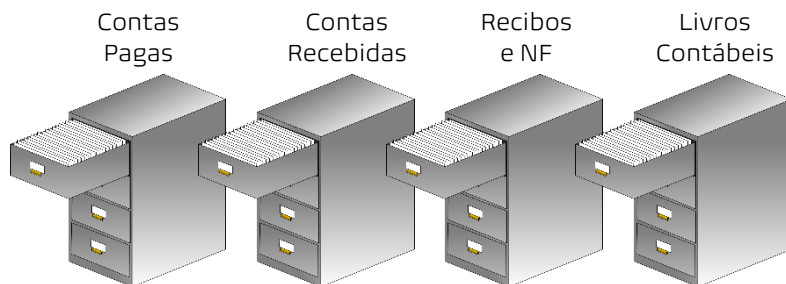
Creio que todos sabemos, que eles surgiram para o armazenamento de documentos e forma organizada. Cada gaveta convenientemente contando com um espaço para identificar seu conteúdo, como por exemplo, a letra inicial do assunto: A, B, C...

Não é à toa que o termo FILE (arquivo) existe em praticamente toda linguagem de programação.

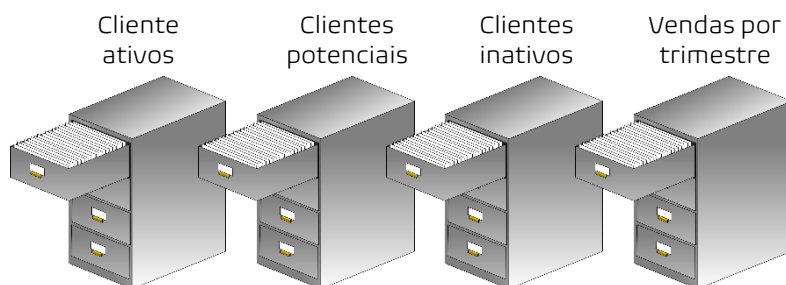
Temos então um arquivo, composto por gavetas onde são armazenadas pastas contendo um ou mais documentos. Cada gaveta possui uma pequena etiqueta, para facilitar pesquisas.

Indo um pouco adiante, imagine uma área financeira de uma empresa. Nessa área são armazenados os documentos de contas pagas, contas recebidas, recibos e notas fiscais e livros contábeis. Para maior praticidade, diferentes arquivos são utilizados para cada assunto, e as gavetas são organizadas por trimestre.

Algo assim:



Agora, imagine que na área de vendas, existem os seguintes arquivos: Clientes ativos, Clientes potenciais, Clientes inativos e Vendas por trimestre.



Dando outro nome a esses componentes, temos os itens básicos de todo banco de dados relacional:

Conjunto dos arquivos de aço	→	Banco de dados
Arquivo de aço	→	Esquema
Gavetas	→	Tabela
Pastas	→	Páginas
Documentos	→	Registros
Etiquetas das Gavetas	→	Índice

Podemos reunir tantos armários de aço quanto necessários. Num banco de dados podemos ter inúmeros esquemas.

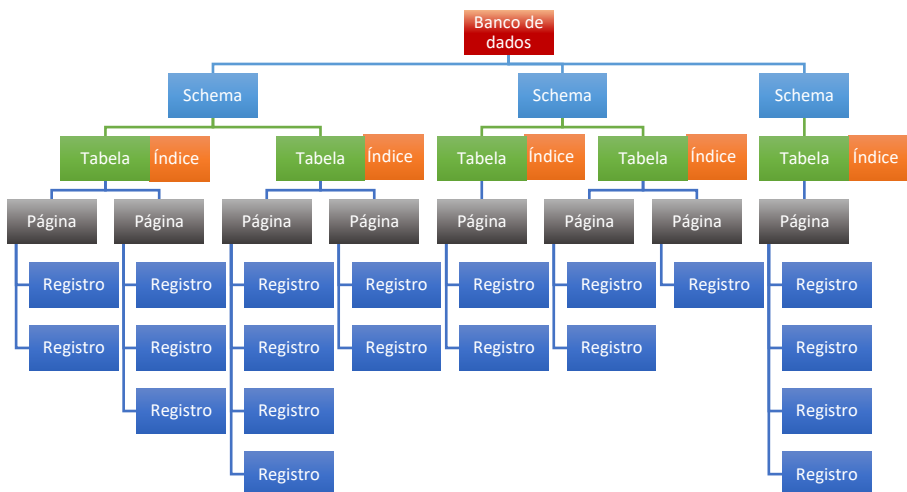
Armários de aço podem ter várias gavetas; do mesmo modo, um esquema pode conter várias tabelas.

Gavetas contêm pastas para reunir documentos similares. Nos bancos de dados, as tabelas são compostas por páginas.

Documentos são a unidade de informação indivisível num armário de aço. Você busca e retira um documento; Não é possível retirar só uma parte dele (não sem danificar 😊). Num banco de dados a unidade indivisível é o registro. Mesmo que você só precise de alguns dados, o SGBD vai resgatar todo o registro para entregar o que pediu.

A para finalizar nossa analogia, os índices de um banco de dados cumprem a tarefa de agilizar pesquisas, tal qual as etiquetas de gavetas.

Graficamente, um banco pode ser visto assim:



Propositalmente citei duas áreas diferentes: financeira e vendas, cada qual com seu conjunto de armários (bancos).

Numa empresa poderíamos encontrar finanças e vendas num **mesmo banco de dados**, em esquemas distintos ou não. Em outra empresa encontramos **bancos separados** para cada área. Não existe certo ou errado.

Agora que já somos experts em armários de aço, podemos fazer uma pausa e esticar as pernas um pouco...

Sistema Gerenciador de Bancos de Dados - SGBD

Em inglês: Relational Database Management System - RDBMS



SGDB = RDBMS

Banco de Dados é uma estrutura para armazenar dados, como visto na analogia. Para construir e gerir essa estrutura é necessário um software: **o gerenciador de bancos de dados**. Portanto:

Banco de Dados

→ Estrutura de dados

Sistema Gerenciador de Banco de Dados

→ Software



É comum encontrar o termo “banco de dados” relacionado aos dois conceitos.

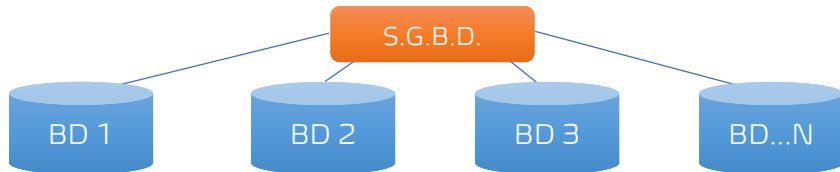
O motivo para esse uso comum do termo é simples. Não existe padrão para a implementação física do banco de dados. Cada fornecedor define uma forma proprietária. Assim, o sistema gerenciador Oracle só é capaz de gerir bancos Oracle, o Postgres só administra bancos Postgres, e assim por diante.

Como o software e a estrutura funciona como uma unidade, o termo “banco de dados” é utilizado de forma genérica.

- “O melhor banco é o Oracle...” → *o melhor sistema gerenciador de bancos de dados*.

- “Nossa contabilidade está no banco de dados MySQL...” → *os dados estão armazenados numa base de dados MySQL*.

Um detalhe importante a conhecer: uma instância de um sistema gerenciador de bancos de dados, ou seja, uma instalação do software executando num servidor qualquer, é capaz de gerenciar mais de um banco de dados.



É totalmente possível encontrar um servidor Microsoft SQL Server gerenciando um banco de dados de um sistema CRM, outro criado para o ERP e ainda outro banco especialista, criado para gestão da linha de produção da empresa.

E para sofisticar ainda mais, dois ou mais sistemas gerenciadores podem "olhar" para a mesma base de dados, servindo dados separadamente. Isso, em particular, veremos mais detalhadamente em capítulos posteriores.

SGDB relacionais mais populares

Existem SGDB gratuitos e pagos. A opção sobre qual utilizar depende de vários fatores, tais como suporte 24x7, capacidade de escalabilidade, sistemas operacionais suportados, volume de dados etc.

Embora exista uma linguagem padrão de consulta, cada SGDB é proprietário. Ou seja, quanto mais utilizado e maior fica um banco de dados, mais complexa é uma eventual troca de fornecedor.



Ranking dos SGDB mais populares

<https://db-engines.com/en/ranking>

Linguagem SQL - surgimento

Um banco de dados que não pode ser consultado é inútil. Logo, assim que surgiram, uma das principais preocupações foi a facilidade de manutenção e consulta. A preocupação motivou várias discussões, e delas surgiu a linguagem SQL (Structured Query Language).

SQL é definida nos padrões ANSI X3.135 de 1986 e adotada pela ISO 9075 de 1987.

Caso queira ler mais a respeito:



<https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/#gref>

Em geral, os padrões definidos são respeitados pelos diferentes fornecedores. Contudo, todos adicionam elementos à linguagem segundo suas necessidades e estratégias.

Assim, é normal inicialmente temer a quantidade de instruções da linguagem. Porém:

- SQL é razoavelmente padronizada e próxima ao inglês;
- O guia de sintaxe específico de cada SGBD estará sempre disponível, via web, com exemplos;
- Existem vários bons editores, que geram os comandos praticamente prontos para uso.



Não se preocupe em memorizar. Einstein não sabia seu número de telefone de cor: - “É para isso que existe a lista telefônica”, dizia ele.

O que vimos nesse capítulo

Bancos de Dados são estruturas para armazenar e gerir grandes quantidades de dados com eficiência.

Bancos de Dados possuem **Esquemas**, que possuem **Tabelas** que armazenam **Registros**.

Índices existem para agilizar pesquisas.

Sistemas Gerenciadores de Bancos de Dados são softwares que efetivamente gerenciam os Bancos de Dados.

SQL é a linguagem padrão para definir e consultar Bancos de Dados.

Muitas vezes a jornada é mais importante que o destino.
Desfrute a jornada.

Capítulo II - Conceitos iniciais, parte II

Já aprendemos os conceitos de Bancos de Dados, Sistemas Gerenciadores de Bancos de Dados e, muito brevemente, o que é a linguagem SQL. Também vimos alguns conceitos adicionais:

Esquema

Schema em inglês

Como vimos anteriormente, esquema é um agrupamento lógico de tabelas. Por exemplo, esquema “de Contas a Receber”.

Mas vamos corrigir para a definição correta:

Um esquema é um agrupamento lógico de diferentes tipos de objetos: tabelas e seus índices relacionados, visões (*views* em inglês), tipos de dados customizados (*data types*), gatilhos (*triggers*), funções (*functions*) e procedures armazenadas (*stored procedures*) e outros.



Veremos os principais tipos de objeto nos próximos capítulos.

Tabela

Table em inglês

Num banco de dados, uma tabela é uma coleção de inúmeros registros de um mesmo tipo, ou seja, que possuem a mesma estrutura e armazenam dados semelhantes entre si.

Nos bancos de dados relacionais **uma tabela é composta por linhas e colunas**. As **linhas** são os registros armazenados enquanto as **colunas** são os dados individuais de cada registro, que denominamos **campos**.

Registro

Num banco de dados relacional um registro é uma linha de dados de uma tabela. Registros de uma tabela possuem sempre os mesmos campos, estejam preenchidos ou não.

Campo

Field em inglês

Cada unidade de informação que armazenamos num registro é um campo. Considere um documento de registro pessoal do Brasil (RG):

- Número de registro geral (RG)
- Nome completo do proprietário do RG
- Data de nascimento
- Sexo
- Nome completo do pai
- Nome completo da mãe
- Nome do órgão emissor do RG
- Data de emissão do RG

... e algumas informações mais.

Cada um desses elementos é um campo.



Um campo armazena uma, e apenas uma ocorrência de um valor.

Isso precisa ficar muito bem compreendido. Um campo sempre armazena apenas um valor. Se, por exemplo, criássemos um campo denominado “Filiação” para registrar o nome do pai e da mãe separados por vírgula, para o SGBD isso seria um dado: filiação. Ele não entende a separação por vírgula.

Tipo de dado

Data type em inglês.

“Data de nascimento” é o mesmo que “Sexo”? Obviamente que não. Sabemos disso intuitivamente. Quando descrevemos um registro para um SGBD precisamos declarar os diferentes campos que ele possui e, o tipo de dado de cada campo.

Assim como só é possível somar células numéricas numa planilha, um SGDB só consegue somar dados numéricos.

Os principais tipos de dados encontrados nos SGBD relacionais são:

varchar ou character varying ou varchar2 ou nvarchar ou national character	Cadeias de caracteres de tamanho variável entre 1 e 8.000. Após a tabela há maiores explicações.
character ou char ou nchar ou national character	Cadeias de caracteres de tamanho fixo, entre 1 e 8.000. Após a tabela há maiores explicações.

int ou integer	Valor numéricos com sinal, armazenados com 4 bytes de comprimento.
bit	Armazenamento dos valores 0 ou 1.
tinyint	Armazenamento de 1 byte de tamanho, com sinal.
smallint	Valores numéricos com sinal, armazenados com 2 bytes de comprimento.
bigint	Valores numéricos com sinal, armazenados com 8 bytes de comprimento.
decimal ou numeric	Valores numéricos com casas decimais. A precisão (quantidade de casas decimais) pode chegar a 17.
float ou double precision	Valores numéricos com sinal, armazenados com 4 ou 8 bytes e podendo ter até 15 casas decimais.
real	Valores numéricos com sinal, armazenados com 4 bytes de comprimento, podendo ter até 7 casas decimais.
date	Armazenamento de datas, sem informação de horas.
time	Armazenamento de horas, sem informação de data.
datetime ou	Armazenamento de data e hora.

timestamp	
text ou cblob	Armazenamento de textos com até 1GB (em geral).
blob ou varbinary	Armazenamento de dados binários, tais como imagens, sons, vídeos etc. Armazenamento até 1GB.
autoincrement	Campos bigint que são automaticamente incrementados em 1 a cada novo registro inserido.

O nome correto a especificar depende do SGBD. Destaquei em azul o nome especificado no padrão ANSI, que deveria ser aceito por todos os fornecedores. Deveria, mas... sempre confirme na documentação do SGBD.

CHAR x NCHAR e VARCHAR e NVARCHAR

O prefixo N refere-se ao padrão Unicode.

O padrão Unicode surgiu para que dados-texto em qualquer idioma fossem suportados. Nesse padrão cada caractere é codificado utilizando 2 bytes. Até então, dados-texto eram compostos de caracteres codificados utilizando 1 byte.

Em geral utiliza-se CHAR e VARCHAR.

CHAR ou VARCHAR?

A diferença entre **char** e **varchar** está na forma como o dado é armazenado no final das contas. **char** sempre armazena a quantidade de dados especificada na definição do campo. Se o dado tem menos caracteres, espaços em branco são adicionados; se possui mais caracteres que o tamanho determinado, o dado é truncado. **varchar** armazena exatamente o tamanho do dado informado. Se a quantidade de caracteres for menor, nenhum espaço é adicionado. Se a quantidade for maior, um erro é disparado e a operação é de inclusão ou atualização é cancelada.

AUTOINCREMENT

Quando avançarmos um pouco mais sobre como os bancos de dados funcionam você notará que ter um campo que é automaticamente incrementado a cada registro é extremamente útil.

Microsoft SQL Server	identity(1,1)
MySQL / Maria DB	auto_increment
Oracle	number generated always as identity(start with 1 increment by 1) <i>Apenas Oracle12c+</i>
Postgres	serial

De fato, Oracle e Postgres não possuem um tipo de dados auto incrementado nativo. Um tipo de objeto denominado "sequence"

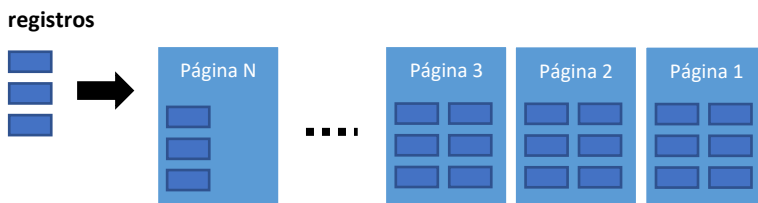
é empregado por trás dos panos. Mas não se preocupe; o resultado é perfeito.

Página

A existência dessa estrutura nem sempre é explicada em cursos sobre bancos de dados. Creio que pelo fato de serem tão automaticamente geridas pelos SGBD, que desenvolvedores e usuários nem as percebem.

Apesar de praticamente nunca lidarmos com elas, é importante sabermos que existem, e como atuam.

Páginas são conjuntos de registros. São criadas pelo SGBD à medida em que são necessárias.



As páginas mais recentes são mantidas em memória, enquanto as mais antigas ficam em disco. Quando consultas são realizadas, o SGBD identifica quais páginas são necessárias, e as trás para a memória. Toda vez que uma consulta é realizada, o SGBD repete o processo, mantendo em memória então, as páginas que são mais acessadas. Quanto mais memória está disponível, mais páginas são mantidas.

Quando registros de uma página são atualizados, o SGBD grava a página em disco, em segundo plano.

Índice

Index em inglês

Em nossa analogia representamos os índices como as etiquetas das gavetas dos armários de aço.

De fato, os índices existem para impulsionar a performance de leitura de um banco de dados relacional.

Um índice é sempre atrelado à uma, e apenas uma tabela e contém uma relação de campos a ordenar, bem como o tipo de ordenação: crescente ou decrescente.

Considere uma tabela contendo todos os moradores da cidade de São Paulo. Como dados, a tabela possui:

- Nome completo
- Idade
- Sexo
- Bairro
- Rua
- N° da casa
- Complemento
- CEP
- Nome social

Qual seria a melhor ordenação dessa tabela?

Depende. Essa é melhor resposta.

Temos que saber quais serão as consultas mais realizadas para conhecer os campos mais importantes.

Se existir uma rotina que conte moradores por idade, um índice contendo apenas "Idade" é o ideal.

Se outra rotina consulta semanalmente a média de idade por bairro, então um índice contendo "bairro" e "idade" é o mais adequado.

De fato, podemos ter tantos índices quanto precisarmos para uma tabela. E isso é muito importante. Não é porque temos "idade" no índice por bairro, que a coluna índice está totalmente ordenada. A ordem das colunas do índice é relevante.

Vamos visualizar isso:

Tabela

1	Lapa	30
2	Centro	59
3	Jabaquara	40
4	Jabaquara	30
5	Lapa	58
6	Lapa	45
7	Centro	30
8	Jabaquara	27

Índice "bairro" e "idade"

7	Centro	30
2	Centro	59
8	Jabaquara	27
4	Jabaquara	30
3	Jabaquara	40
1	Lapa	30
6	Lapa	45
5	Lapa	58

O número é a apenas a sequência de armazenamento.

Note que no índice as idades não estão ordenadas de forma crescente, quando a observamos sozinha. Ela é crescente por bairro.

Chave Primária

Primary Key em inglês

Um tipo de índice é particularmente importante em bancos de dados relacionais. O índice de chaves primárias.

Uma tabela pode funcionar como uma lista simples, onde valores repetidos são aceitos. Por exemplo:

Carro

Gol

Celta

Voyage

Celta

Toro

Uno

Polo

Celta

Essa poderia ser uma tabela que recebeu o nome do primeiro carro que os alunos de uma turma digitaram a pedido do professor. Note que “Celta” repete-se três vezes.

Agora, imagine que o aluno que digitou o primeiro Celta na verdade viu um Onyx e pede para o professor corrigir. Como o professor diria para o SGBD “altere a primeira ocorrência de ‘Celta’ para ‘Onyx’”?

Poder referir-se a um registro específico de uma tabela é fundamental, mesmo nas tabelas mais simples. A essa capacidade denominamos **chave primária**. Pode ser o valor de uma única coluna, ou um conjunto de várias colunas.

Para nossa tabela de idades por bairro, por exemplo, “Nome completo”, “CEP”, “Nº da casa” e “Complemento” formariam uma chave primária relativamente segura.

Já para nossa tabela de carros teríamos que adicionar uma nova coluna. Uma coluna do tipo “autoincremento”, como vimos anteriormente.

Como boa prática, empregar uma coluna autoincrementada é preferível a definir índices de múltiplas colunas.

Relacionamento

Como explicado anteriormente, campos armazenam apenas uma instância de um valor qualquer. Já sabemos que uma tabela é composta por campos, e todos os registros dessa tabela possuem os mesmos campos.

Como seria então, uma tabela para representar uma Nota Fiscal?

Uma NF possui um corpo, com dados gerais tais como N° da nota, data de emissão, valor total etc. e um tabela de itens para um ou mais ocorrências: código do item, descrição, quantidade, valor unitário, descontos, impostos, valor total, por exemplo.

Uma alternativa seria criar campos “cod_item_1”, “cod_item_2”, “cod_item_3”; e perguntar ao responsável, qual é o máximo de itens possível, para limitar as repetições.

Além de trabalhoso, isso tornaria a solução específica para o tipo de NF avaliada. E, se no futuro o modelo de nota for

alterado para suportar mais linhas, será necessário adicionar mais colunas na tabela, além de eventualmente alterar sistemas para utilizar essas novas colunas.

A solução ideal é armazenar os dados gerais da NF numa tabela, e os itens de todas as NF em outra tabela, vinculando as duas de alguma forma.

Já vimos o conceito de chave primária, e que é possível utilizar mais de um campo para defini-las.

Então, algo assim resolveria:

<u>NF</u>	<u>Itens NF</u>
Nº da NF (chave primária)	Nº da NF (chave primária)
Data de emissão	Nº do item (chave primária)
Valor total	Código do produto
Total de impostos	Descrição
...	Valor unitário
	Valor dos descontos
	Valor dos impostos
	Valor total do item

Agora, como dizemos para o SGBD, que “Itens NF” relaciona-se com “Tabela NF”, e como essa relação funciona?

Chave Estrangeira

Foreign Key em inglês

Para criar uma relação entre duas tabelas num banco de dados relacional, definimos chaves estrangeiras. Para definir uma chave estrangeira, seguimos os seguintes passos:

- Defina qual é a tabela-pai e qual é a tabela-filha;
- Defina os campos-chave em cada tabela;
- Defina a cardinalidade da relação;
- Defina o que acontece quando o registro da tabela-pai é alterado ou excluído.

No nosso exemplo de uma Nota Fiscal:

- 1) Tabela-pai: "NF"
Tabela-filha: "Itens NF"
- 2) NF: campo "Nº da NF"
Itens NF: campo "Nº da NF"
- 3) Relação 1 x N, ou seja, uma NF pode conter vários itens.
- 4) Alteração em NF: Propagar para tabela "Itens NF"
Exclusão de NF: Propagar para tabela "Itens NF"

Desse modo, as tabelas "NF" e "Item NF" estariam relacionadas pelo campo "Nº da NF"; sendo que uma nota pode relacionar-se um zero ou mais itens de NF. Se ocorrer uma alteração do campo "Nº da NF" na tabela NF, o SGBD automaticamente atualizará a tabela "Itens NF" (coluna Nº da NF) para manter a consistência. E se um registro for excluído na tabela NF, o SGBD excluirá automaticamente todos os itens relacionados.

Alteração e Exclusão com chaves estrangeiras

“Defina o que acontece quando o registro da tabela-pai é alterado ou excluído”.

Quando existem relações, decidir o que deve ocorrer quando o registro da tabela-pai é alterado ou excluído é importante.

Primeiro, entenda que, se o registro da tabela-pai foi alterado, mas nada mudou nos campos utilizados na relação com a tabela filha, nenhuma ação será tomada pelo SGBD. O problema é quando um campo-chave tem seu valor alterado. Nesse breve instante em que o valor muda, perdemos a relação com os registros da tabela-filha, que ainda não foi atualizada.

Excluir o registro da tabela-pai é o mesmo que alterar todos os campos-chave. Perde-se a relação com os registros da tabela-filha naquele momento.

Há três ações possíveis a configurar para essas situações:

Propagar (*Cascade em inglês*)

Alteração: O novo valor dos campos-chave é copiado para os campos-chave da tabela-filha.

Exclusão: Os registros da tabela-filha são todos excluídos.

Restringir (*Restrict em inglês*)

Se há registros na tabela-filha a atualização da tabela-pai é abortada e o SGBD gera uma mensagem de erro, tanto para operações de alteração quanto exclusão.

Preencher nulo (*SET NULL em inglês*)

Os campos-chave da tabela-filha são preenchidos com NULL, tanto nas operações de alteração quanto exclusão.

Logo, quem define o banco de dados é quem configura como a integridade dos relacionamentos será mantida, caso-a-caso.

E novamente, não existe certo ou errado. Tudo depende de decisões de gestão.

Pode parecer natural excluir todos os itens de uma NF que for excluída. Mas poderia existir uma regra, que NF só podem ser excluídas enquanto não houver itens lançados. Nesse cenário, empregar "RESTRICT" ao invés de "CASCADE" seria o correto.



Isso será exemplificado para compreender na prática.

Chave Única

Unique Key em inglês

Bem, já entendemos a importância das chaves primárias num banco de dados relacional.

Mas, como vimos anteriormente, podemos definir vários índices para uma tabela. Tantos quantos necessitarmos, para otimizar as inúmeras consultas que serão feitas.

Outro recurso importante que os SGBD relacionais fornecem são as chamadas chaves únicas. São índices que garantem que o dado de uma coluna ou conjunto de colunas não se repete na tabela, ainda que não representem a chave primária da tabela.

Onde usamos isso?

Deixe-me dar um exemplo:



Numa declaração de imposto de renda o CPF do declarante é chave primária. Há uma tabela de dependentes, onde, claramente, a chave primária é o CPF do declarante + id autoincrementado. Isso porque, um dependente pode ser um filho que ainda não tenha um CPF, e não é razoável assumir que o nome dos dependentes não se repita. O melhor que se pode fazer, é evitar o erro. Então, define-se que "CPF", "nome" e "grau de parentesco" seja uma chave única da tabela de dependentes.

Chave primária Declaração:

"CPF declarante"

Chave primária Dependentes:

"CPF declarante"

"Sequencial"

Chave única Dependentes:

"CPF declarante"

"CPF dependente"

"Nome dependente"

"Grau de parentesco"

Uma família com uma mãe chamada Maria Flor e uma filha Maria Flor não seria um problema, já que o grau de parentesco muda, e potencialmente teremos CPF diferentes.

Já uma família com dois filhos chamados "João Júnior" e sem CPF seria complicado. Complicado até para eles...

O que vimos nesse capítulo

Revisamos os conceitos de **Esquemas, Páginas, Tabelas, Registros e Índices**.

Aprendemos sobre **Campos e Tipos de dados**.

Fixamos o conceito de que um **campo só armazena uma instância de um valor**.

Conhecemos os principais tipos de dados existentes nos bancos mais populares, incluindo o super útil **autoincrementado**.

Apresentamos os conceitos de **Chave Primária, Relações, Chaves Estrangeiras e Chaves Únicas**.

Vimos como determinar **ações numa relação**, quando o registro da tabela-pai é **alterado** ou **excluído**.

O alicerce não é vistoso, mas sustenta o edifício que é notado.

Capítulo III – Primeiros passos práticos

De mestres dos arquivos-de-aço a detentores do conhecimento, dois passos muito produtivos!

Temos vários conceitos por ver ainda, mas já é hora de colocar a mão na massa. Ninguém vence 100m sem treinar, treinar muito.

Poderíamos utilizar uma versão gratuita do Oracle ou MS SQL Server. Postgres também seria uma boa opção. Mas optei aqui, por utilizarmos MySQL. Elenco minhas razões:

- MySQL é grátis, e muito fácil de instalar em qualquer S.O.;
- Contém todos os tipos de objeto que vamos estudar;
- Conta com um utilitário de administração muito bom e leve, comparado com os concorrentes (Workbench).

Mas, se em algum ponto considerar relevante explicar como algum conceito é empregado nos outros SGBD citados, eu o farei.

Para iniciar vamos instalar os dois programas.

Instalação do SGBD MySQL Server

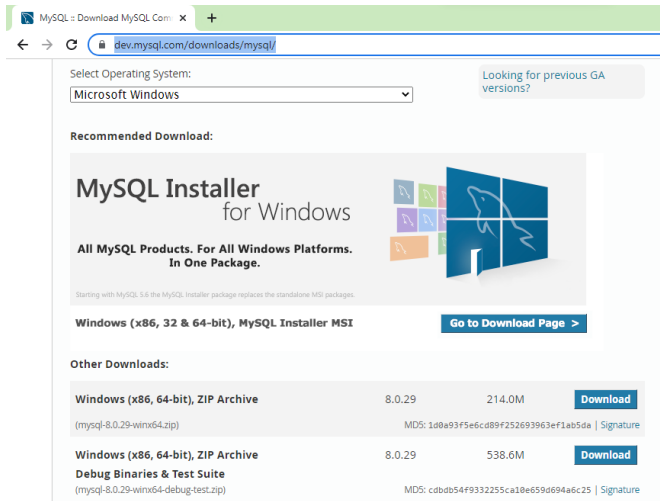
Se você já possui um servidor local, ou em nuvem, poderá utilizá-lo. Nenhuma versão específica é necessária para os exemplos e exercícios desse pequeno guia.

Para instalar, acesse o link de download da versão Community, que é gratuita.



<https://dev.mysql.com/downloads/mysql/>

Se você é usuário de Windows, a versão recomendada pelo site é a ideal.



Ao clicar em “Go to Download Page”, você será direcionado para uma página com duas opções:

MySQL Installer 8.0.29

Select Operating System: Looking for previous GA versions?

Windows (x86, 32-bit), MSI Installer (mysql-installer-web-community-8.0.29.0.msi)	8.0.29	2.3M	Download
		MD5: 4f735569267527dec28d9e8d977f33d1 Signature	
Windows (x86, 32-bit), MSI Installer (mysql-installer-community-8.0.29.0.msi)	8.0.29	439.6M	Download
		MD5: 3f4def7aef75e2e038e2dd62e784f246 Signature	

We suggest that you use the [MD5 checksums](#) and [GnuPG signatures](#) to verify the integrity of the packages you download.

A primeira baixa o instalador, e após realiza downloads da web durante a instalação. A segunda baixa todos os pacotes, para uma instalação *offline* posterior.

Escolha a que preferir. Aguarde o download, e faça a instalação padrão.

Instalação do utilitário MySQL Workbench

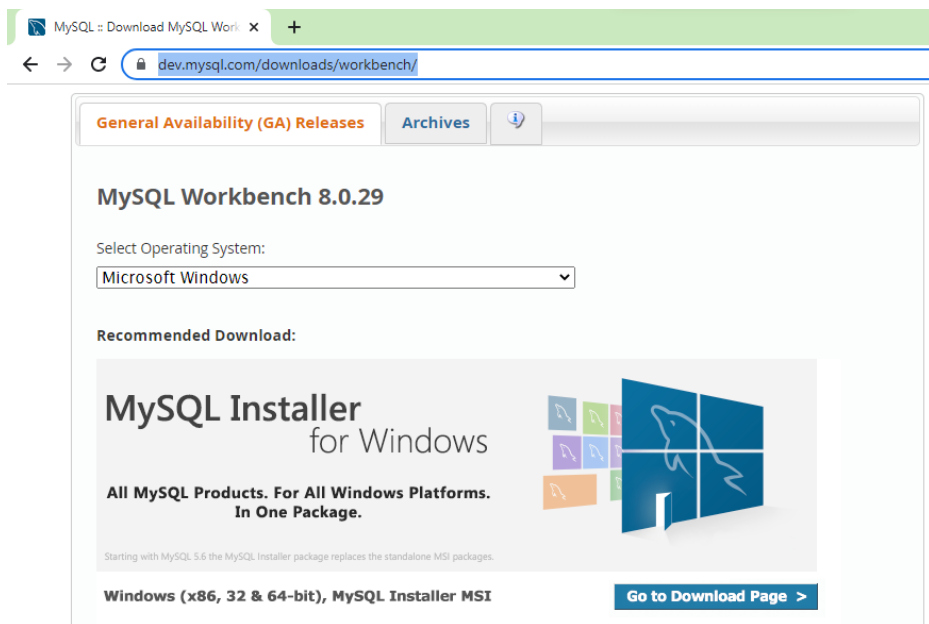
Para interagir com o MySQL utilizaremos uma ferramenta visual. Isso ajuda muito.

Se já a possui, ignore esse passo.

Para instalar, obtenha o programa do link oficial:

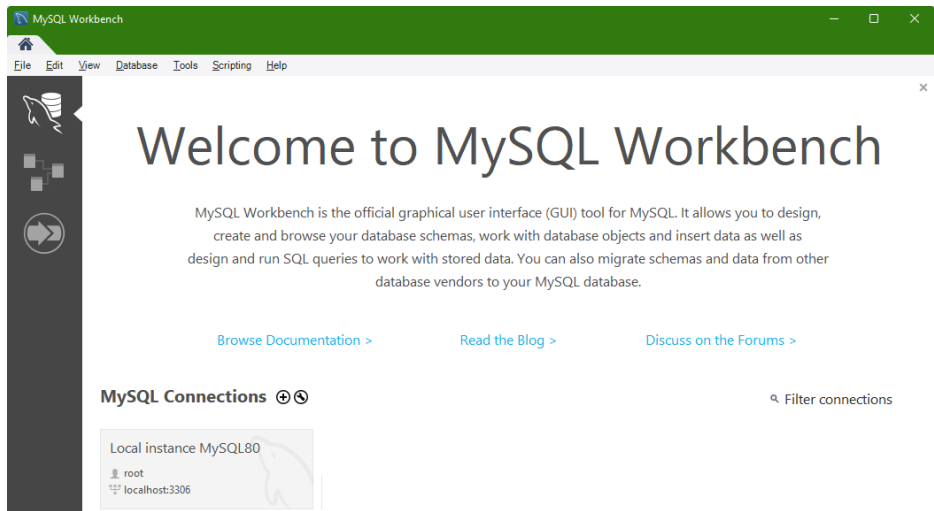


<https://dev.mysql.com/downloads/workbench/>



Novamente, uma tela com duas opções é apresentada; a que faz downloads dos pacotes durante a instalação e a que baixa tudo de uma vez. Prossiga como preferir e realize a instalação padrão.

Concluídas as duas instalações, ao executar o Workbench, uma tela semelhante a essa deverá ser apresentada:



Uma conexão com o servidor MySQL local já será apresentada (**Local instance MySQL80**), ou algo similar.

Abaixo da descrição da conexão está o nome de usuário MySQL empregado na conexão. Isso pode variar de instalação para instalação, mas em geral é o usuário **root**.

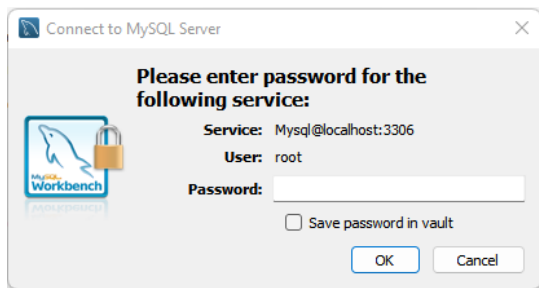
Abaixo do usuário está a identificação do servidor. Em geral, **localhost:3306**.

Localhost é o nome do computador local, que mapeia para o IP 127.0.0.1. Tanto no Windows, quanto Linux.

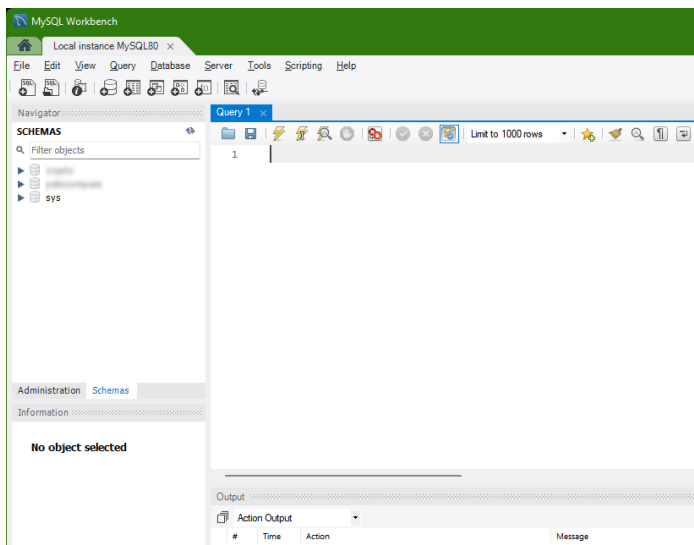
3306 é a porta TCP/IP padrão do MySQL.

Certamente esses dados podem ser diferentes em sua máquina, e se tudo funcionar quando você clicar nesse box, tudo bem.

Quando clicamos no box podemos ir diretamente para a tela que apresenta o conteúdo do banco de dados, ou sermos indagados pela senha do usuário, como é o meu caso:



Basta digitar a senha do usuário, como definida na instalação do MySQL; correndo tudo bem, veremos algo assim:



No painel esquerdo temos os Esquemas existentes no banco de dados local e à direita um editor para editarmos nossas instruções SQL.

Caso tenha qualquer dificuldade em chegar aqui, entre em contato via WhatsApp, e combinamos um dia e horário para fazemos juntos.



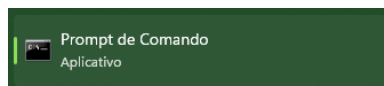
(11) 99729-0935

Criação do Banco de Dados

A primeira ação que realizaremos será criar um banco de dados exclusivo para os exercícios desse guia.

Para essa tarefa vamos utilizar tanto o utilitário de linha de comando **mysql** quanto o Workbench. Isso apenas para enriquecer seu portfólio de ferramentas. Não iremos ficar realizando coisas na linha de comando, não se preocupe.

Abra uma sessão do Prompt de comandos (shell).



Na tela que aparecerá digite: **mysql -u root -p**

```
PS C:\> mysql -u root -p
Enter password: |
```

A senha será solicitada. Informe e pressione Enter.

Se correr tudo bem, teremos a seguinte tela:

```
PS C:\> mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 46
Server version: 8.0.27 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> |
```

Estamos agora no utilitário de linha de comando para interação com o MySQL.

Instruções SQL devem terminar com ponto-e-vírgula (;).

Vamos criar um banco de dados denominado **db_a_apagar**.

Digite: **create database db_a_apagar;**

Pressione Enter.

Deveremos ter o seguinte resultado:

```
mysql> create database db_a_apagar;
Query OK, 1 row affected (0.38 sec)

mysql> |
```

Caramba, você acaba de criar um banco de dados!

Isso mesmo. Tão simples quanto isso.

Mas vamos apagar esse banco. Foi apenas para um primeiro contato com o utilitário de linha de comando.

Agora, digite: **drop database db_a_apagar;**

Pressione Enter.

```
mysql> drop database db_a_apagar;  
Query OK, 0 rows affected (0.56 sec)
```

Pronto, o banco foi apagado.

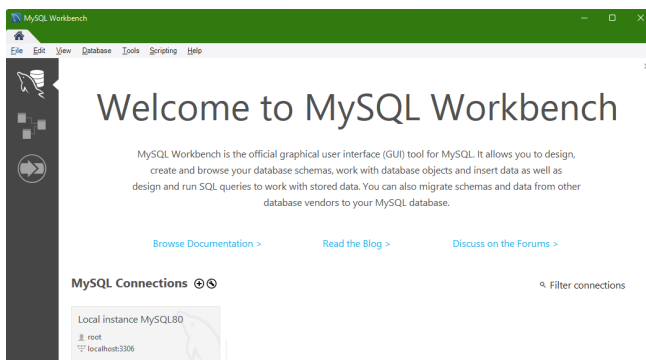


Apesar de existirem proteções, perceba o quão perigoso é ter uma conta com plenos direitos sobre os bancos. Uma pessoa com más intenções pode provocar um grande prejuízo. Pensando nisso, além dos controles de sistema operacional usuais (contas de usuário), os SGBD possuem recursos para limitar o que cada usuário pode ou não fazer. Não vimos isso ainda. Por agora, entenda a importância deles.

Digite **exit** e pressione Enter para sair do utilitário **mysql**.

Agora digite **exit** e pressione Enter para fechar o shell.

Voltemos ao Workbench. Especificamente, para a tela principal:



Clique na instância "Local instance MySQL80" para abrir uma sessão com o banco principal.

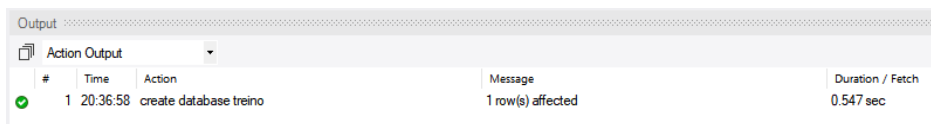
Agora, na parte esquerda (área para instruções SQL), digite:

create database treino;

Tecle Enter ao final da linha.

Para executar a instrução, pressione Ctrl + Enter.

Na parte inferior da tela, deve aparecer:



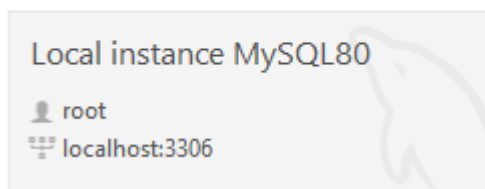
The screenshot shows the 'Output' window with a tab labeled 'Action Output'. It contains a table with the following data:

#	Time	Action	Message	Duration / Fetch
1	20:36:58	create database treino	1 row(s) affected	0.547 sec

Feche a aba, voltado para a tela inicial.

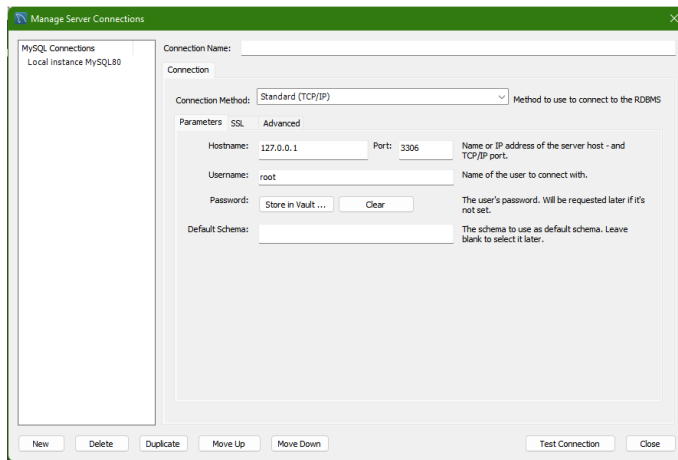
Agora, clique no ícone de ferramenta que aparece à direita de MySQL Connections:

MySQL Connections ⊕ ⊞



Será apresentado um diálogo listando as conexões existentes, uma série de dados e botões na parte inferior.

Clique no botão "New":



Quando clicar em "New" o campo "Connection Name" mudará para "new connection", e os demais dados serão reinicializados.

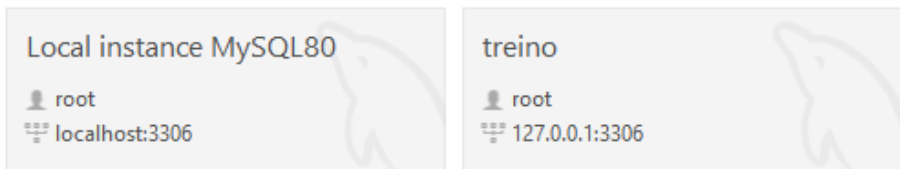
Alterar "new connection" para **treino**.

No campo Default Schema preencha **treino**.

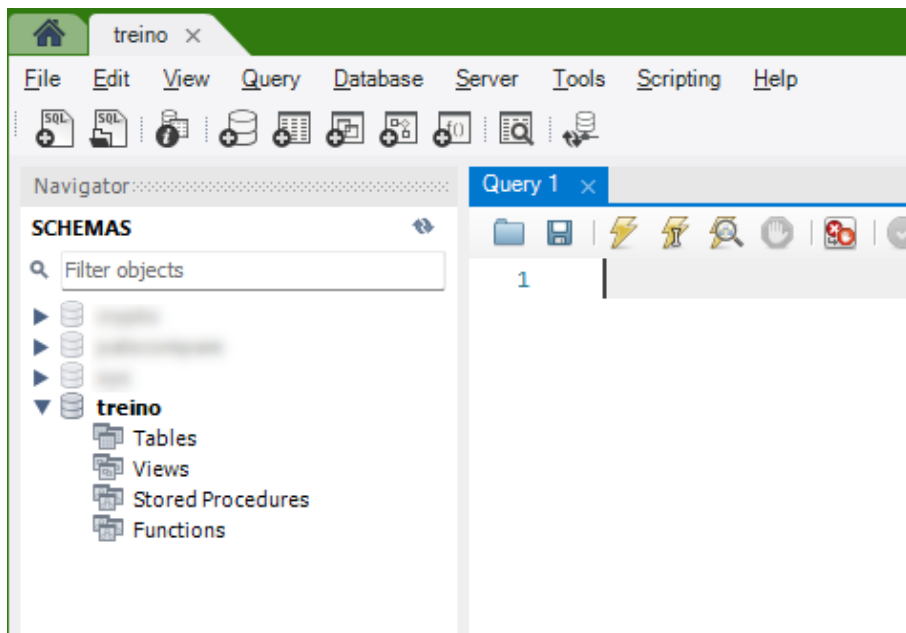
Clique no botão Close.

Deveremos ver agora, uma nova conexão "treino" na tela principal:

MySQL Connections



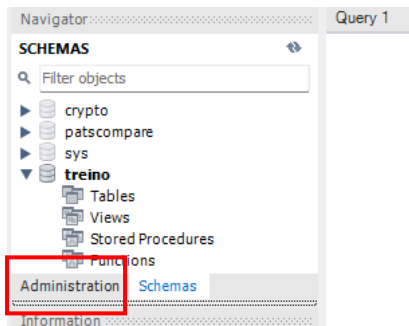
Clicando na conexão e informando a senha, deveremos ver algo assim:



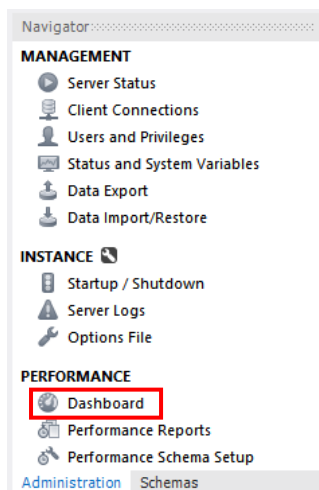
Sucesso! Criamos e conectamos com nosso banco de dados.

Note que o SGBD gerou automaticamente um Esquema com o mesmo nome do banco de dados. Isso é normal com MySQL.

Apenas a título de curiosidade, selecione a aba "Administration" que fica à direita, logo abaixo da lista "Schemas".



No menu que aparecerá, clique em Dashboard.



Propositalmente, não mostro aqui o resultado. Mas você perceberá que bacana. Lembre-se que tudo aqui é gratuito.

Definição do usuário de acesso

Agora, vamos adicionar uma pitada de segurança em nosso ambiente de treinamento.

Em ambientes profissionais não teremos uma conta com total acesso, como tivemos até aqui. De fato, será tarefa de um DBA criar o banco e nos disponibilizar. Logo, estamos aqui exercitando algumas funções do DBA (*DataBase Administrator*).

Na janela de scripts, digite e execute os seguintes comandos SQL:

```
create user 'usr_treino'@'localhost'  
identified by 'treino';
```

```
grant all on treino to 'usr_treino'@'localhost';
```

Output			
Action Output			
#	Time	Action	Message
✓ 1	21:00:50	create user 'usr_treino'@'localhost' identified by 'treino'	0 row(s) affected
✓ 2	21:00:52	grant all on treino to 'usr_treino'@'localhost'	0 row(s) affected

Feche a aba de scripts para retornar à tela principal.

Agora, clique no ícone de ferramenta para que o diálogo de manutenção de conexões apareça.

MySQL Connections

Clique na conexão **treino** que aparece à direita.

Altere o campo Username para **usr_treino**

Clique no botão **Store in Vault ...**

Digite **treino** e clique em Ok.

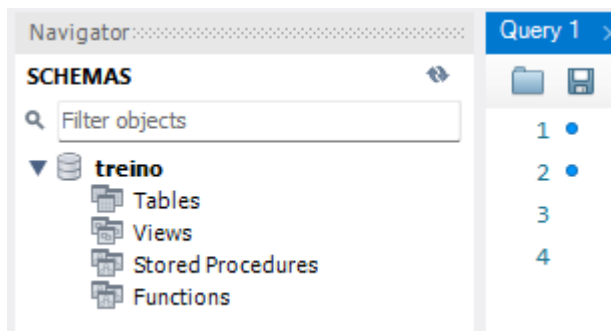
Clique em Close para fechar o diálogo.

Clique na conexão **treino**:

MySQL Connections

Local instance MySQL80 root localhost:3306	treino usr_treino 127.0.0.1:3306
--	---

Agora não será solicitada senha, e somente o Schema **treino** aparecerá à esquerda.



O que fizemos aqui foi criar uma conta de usuário MySQL denominada **usr_treino** que só tem privilégios no esquema **treino** do banco de dados **treino**.

Todas essas etapas, de fato, competem a um DBA, mas agora você já sabe como ele faz.

Criação de Schemas

Como visto anteriormente, um banco de dados pode possuir vários esquemas. Um esquema contém diferentes objetos.

Para compreender o uso desses esquemas, vamos adicionar mais um, denominado **treino_prod**, abreviação de “treino produção”.

Em alguns SGBD, o DBA executaria os seguintes comandos:

```
create schema treino_prod;
```

Mas o MySQL em particular, vincula cada schema a um banco de dados dedicado. Ou seja, quando emitimos esses comandos, ele tenta criar um banco de dados com o mesmo nome do esquema. E como limitamos os privilégios do nosso usuário **usr_treino**, isso falhará.

👉 Essa vinculação um-para-um é específica do MySQL.

O efeito prático, entretanto, é o mesmo para nós. Só temos que executar a criação do novo esquema utilizando a conta de DBA (root).

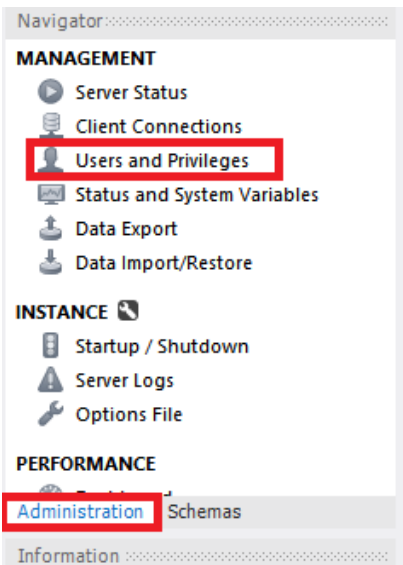
Volte para a tela inicial, onde temos as conexões, e clique na conexão “Local instance MySQL80”.

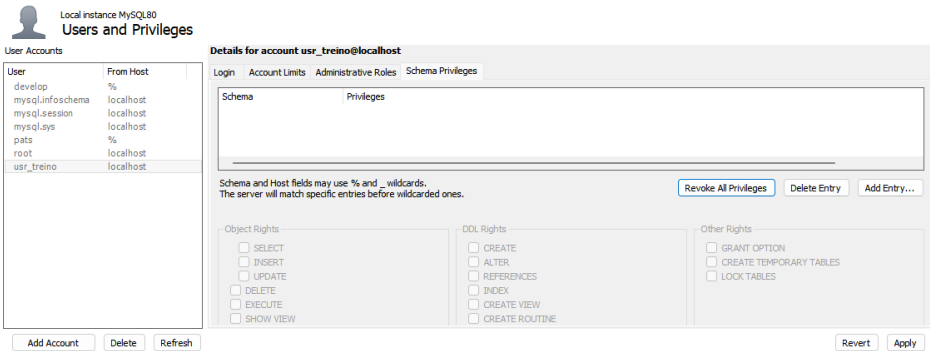
A senha novamente é requisitada. Informe-a para completar o acesso como DBA.

Agora, execute o seguinte comando:

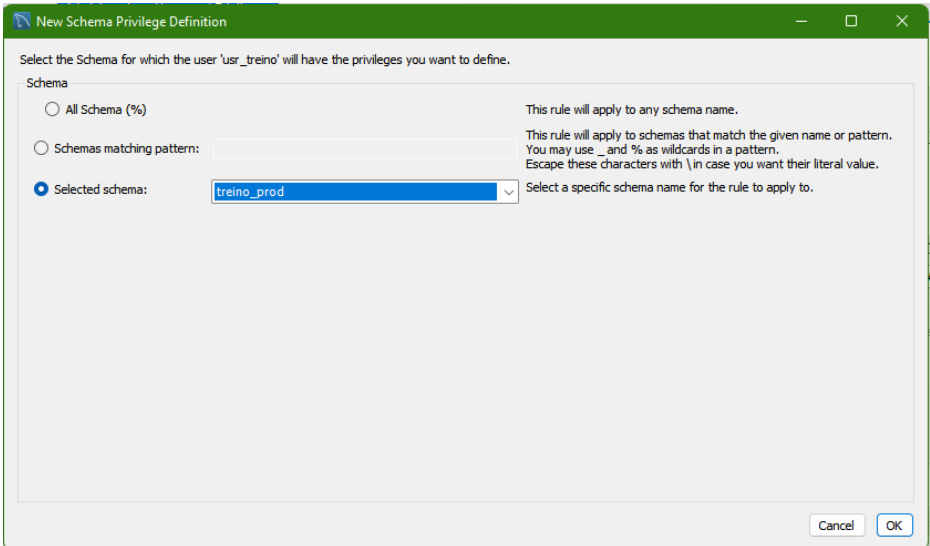
```
create schema treino_prod;
```

Clique na aba “Administration” e em seguida, em “Users and Privileges”:

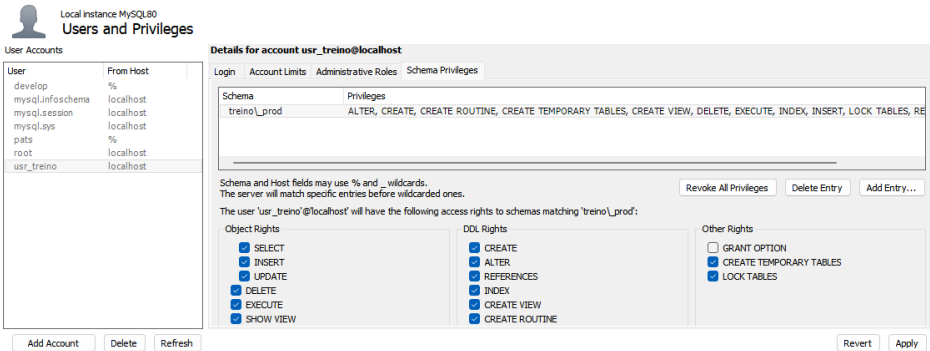




Selecione o usuário **usr_treino** e clique no botão **Add Entry...**



Selecione o esquema **treino_prod** como demonstrado na figura e clique em OK.

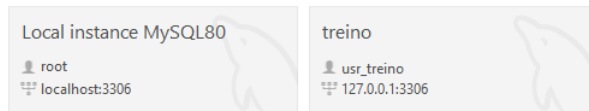


Marque todas as opções, exceto “GRANT OPTION”, e em seguida clique “Apply”.

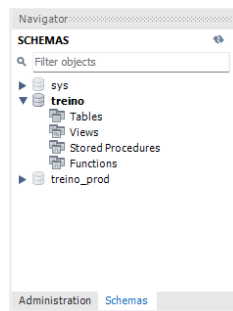
Repita o mesmo procedimento para o esquema **treino**.

Feche a aba para voltar à tela inicial, e nela, clique novamente na conexão **treino**.

MySQL Connections



Clique na aba “Schemas”, caso “Administration” esteja ativa.



Note que agora temos **treino** e **treino_prod** visíveis.

Para fechar com chave de outro esse capítulo, vamos criar um usuário “apenas leitura” para nossos esquemas.

Criação de usuários adicionais

Para criar nosso usuário “somente leitura”, volte à tela inicial, e acesse com o login de DBA – conexão (“Local instance MySQL80”).

Na console de script, execute os seguintes comandos:

```
create user tst_treino;  
grant select on treino.* to tst_treino;  
grant select on treino_prod.* to tst_treino;  
commit;
```

Há um bug no Workbench. Mesmo utilizando opções “Refresh all”, não vemos essas modificações de imediato.

Feche e reabra o programa novamente.

Conectando por "Local instance MySQL80" e selecionando a aba "Administration", item "Users and Privileges", confirmamos a operação:

Navigator

MANAGEMENT

Server Status

Client Connections

Users and Privileges

Status and System Variables

Data Export

Data Import/Restore

INSTANCE

Startup / Shutdown

Server Logs

Options File

PERFORMANCE

Administration

Schemas

Query 1

Administration - Users and Privil...

Local instance MySQL80

Users and Privileges

User Accounts

User	From Host
develop	%
mysql.infoschema	localhost
mysql.session	localhost
mysql.sys	localhost
pats	%
root	localhost
tst_treino	%
usr_treino	localhost

Details for account tst_treino@%

Login

Account Limits

Administrative Roles

Schema Privileges

Schema	Privileges
treino_prod	SELECT
treino	SELECT

Schema and Host fields may use % and _ wildcards.
The server will match specific entries before wildcarded ones.

Object Rights

DQL Right

O que vimos nesse capítulo

Realizamos a **instalação do SGBD MySQL Server**.

Realizamos a **instalação** do software **MySQL Workbench**.

Conhecemos o **utilitário de linha de comando mysql**, e como realizamos login, **criamos e apagamos um banco** de dados com essa ferramenta.

Criamos um banco de dados com o Workbench.

Aprendemos a configurar uma conexão com o novo banco de dados no Workbench, que **não pede senha a toda hora**.

Vimos a ferramenta **Dashboard** do Workbench, que reúne várias estatísticas de performance do SGBD.

Aprendemos a **criar um usuário** específico para o novo banco de dados criado, aumentando a segurança do ambiente.

Aprendemos a **criar esquemas (schemas) adicionais**, e como habilitar acesso a elas para usuários existentes no SGBD.

Finalmente, criamos **um usuário** que só tem direito de consultar os esquemas criados, **sem permissão para alterar nada**.

“Então Deus disse: — Que haja luz! E a luz começou a existir...”

Gênesis 1:1

Capítulo IV – Criação de Tabelas

Até aqui estudamos a teoria de bancos de dados e realizamos a parte prática que podemos chamar de burocrática, de inicialização de um banco.

Criar um banco, definir alguns usuários e configurar permissões são tarefas relativamente simples.

Claro, não estamos falando aqui, de um banco de dados global, com milhões de acessos simultâneos. Mas essa base é bastante adequada para grande parte de soluções reais.

Agora chegou a hora em que planejamento e análise são essenciais. Chegou a hora de definir as tabelas, índices e relacionamentos que teremos.

Logicamente, você terá uma tarefa facilitada aqui, uma vez que as soluções estarão no guia. Contudo, buscando proporcionar a experiência que um desenvolvedor vivencia nesse estágio, seguirei os seguintes passos:

- a. Apresentarei alguns detalhes SQL adicionais necessários no processo de criação de tabelas e seu relacionamento;
- b. Darei um exemplo fora de contexto, apenas para fixar a ideia de uso desses elementos adicionais;
- c. Farei a proposição de quais tabelas precisaremos definir;
- d. Apresentarei minha solução para as tabelas propostas.

Sugiro que, ao chegar ao passo C você faça uma pausa, e tente criar as tabelas sem consultar a solução. Só após, passe ao último passo e compare.

Não existe apenas uma resposta, é fato. Mas agindo assim, será mais simples identificar pequenos pontos eventuais equívocos, compreender “por que isso e não aquilo...”; e desse modo, penso eu, a fixação de conceitos será privilegiada.

SQL e criação de tabelas

Para criar tabelas utilizamos a instrução **CREATE TABLE**. A sintaxe dessa instrução é complexa, e deve ser aprendida em partes. Numa visão geral é:

```
CREATE TABLE (IF NOT EXISTS) nome_tabela (  
    Definição de campo ou restrição 1,  
    Definição de campo ou restrição 2,  
    ...  
    Definição de campo ou restrição N  
);
```

IF NOT EXISTS é uma especificação opcional, que diz ao SGBD que nada deve ser feito caso a tabela já exista. O padrão é o SGBD gerar uma mensagem de erro e parar de executar instruções, caso encontre uma instrução CREATE TABLE para uma tabela que já exista.

No modo interativo, onde estamos executando comandos manualmente, tudo bem receber um erro. Mas num script de inicialização, que queremos que seja totalmente executado para preparar um banco qualquer, isso é inconveniente. Daí a necessidade e conveniência de “IF NOT EXISTS”.

Em seguida vem o nome da tabela. Deve ser um nome único, sem espaços e caracteres especiais, tipo @#\$%&* e outros. O comprimento máximo e caracteres que podem ser utilizados dependem de SGBD para SGBD.

Ainda, o nome da tabela deve ser único no esquema a qual pertence. Se já temos uma tabela “CLIENTE” em nosso esquema, não podemos definir outra. Agora, se há essa tabela em outro esquema, tudo bem. O SGBD permite e sabe como administrar isso.

Após o nome da tabela vêm as definições de campos e restrições (*constraints em inglês*), que devem ser envolvidas por parêntesis e separadas umas das outras por vírgula (,).

Definições de campos

A definição de um campo de tabela tem a seguinte estrutura:

```
nome_campo tipo_dado [(tamanho [,decimais [parâmetros  
específicos]])] [[not] null] [default [valor_padrao]]  
[unique]] [atributos_especificos_SGBD] [,]
```

Os parêntesis aqui representam partes opcionais. Não devem aparecer na instrução.

Complicado? Nem tanto...

Vamos exercitar com um caso real, e essa sopa de parêntesis vai ficar mais fácil de compreender.

Vamos codificar a instrução para criar a seguinte tabela:

Campo	Tipo	Tam	Dec	Obrig?	Valor padrão	Detalhes
ID	Inteiro	-	-	Sim	-	Autoinc.
Nome	Alfanum.	50	-	Sim	-	
Sexo	Alfabético	1	-	Sim	-	M ou F
Nascto	Data	-	-	Sim	-	
Cidade Natal	Alfanum.	80	-	Não	N/I	
CPF	Inteiro	11	-	Sim	-	Único na tabela
Salario	Numérico	7	2	Não	0,00	> ou = 0,00

Chamaremos nossa tabela de FUNCIONARIO.

A instrução para criar essa tabela pode ser:

```
CREATE TABLE IF NOT EXISTS funcionario (  
    id            int            auto_increment primary key,  
    nome          varchar(50)    not null,  
    sexo          char(1)        not null,  
    nascto        date           not null,  
    cid_natal     varchar(80)    default 'N/I',  
    cpf           decimal(11,0)  not null unique,  
    salario       decimal(7,2)   default 0.00  
);
```

Tome muito cuidado com o esquema ativo antes de executar o comando. Caso a tabela acabe sendo criada no esquema errado, execute o comando **DROP TABLE funcionario;** e tente novamente.

No primeiro campo (id) temos isso: [atributos_especificos_SGBD]

“**auto_increment primary key**” é específico do MySQL, para especificarmos uma chave primária autoincrementada.

Após o nome de cada campo temos o tipo de dado. Onde necessário, entre parêntesis, especificamos o tamanho do campo (ou tamanho máximo); e onde há necessidade de especificar a quantidade de casas decimais, isso está presente.

O termo **not null** é empregado para especificar os campos obrigatórios. Não está presente na chave primária, mas lá isso é implícito.

O termo **default** é utilizado para fornecermos o valor padrão, quando existe. Repare que, sendo um valor para um campo

varchar, *char*, *date* ou *datetime*, utilizamos aspas simples ou duplas.



O uso de aspas simples ou duplas pode não ser aplicável a todos SGBD. Inclusive, pode existir suporte para outros delimitadores. MS SQL Server, por exemplo, aceita colchetes em algumas situações. Sempre verifique a documentação do SGBD que precisar utilizar.

Empregamos **unique** para informar um campo que não pode receber um valor repetido (**cpf**).

E concluímos fechando parêntesis e ponto-e-vírgula.



Instruções SQL são *case insensitive*. Maiúsculas ou minúsculas podem ser empregadas livremente. Só tome cuidado com valores-padrão e outros termos que definimos entre aspas simples. Nesse caso são dados, e devemos considerar como os desejamos (maiúsculas, minúsculas, primeira maiúscula etc.).

Se você avaliar a especificação textual da tabela desejada, e comparar com a instrução SQL, notará que pelo menos três pontos não foram atendidos:

- O campo "Sexo" só deve aceitar os valores M ou F.
- O CPF tem que conter 11 dígitos numéricos, e não deve aceitar letras, espaços ou caracteres especiais.
- O valor de salário não pode ser negativo.

Restrições

Constraints em inglês

Para conseguir implementar essas tratativas precisamos recorrer a outro recurso dos bancos de dados relacionais: as restrições, ou *constraints* como são mais conhecidas.

SQL define o seguinte conjunto de *constraints*:

Primary key

As chaves primárias, que impõem que não existam registros duplicados numa tabela, e impõem que os campos informados não possuam o valor NULL.

Foreign key

Utilizada para especificarmos uma chave estrangeira, que relaciona duas tabelas; Determina também, o que ocorre na alteração e exclusão do registro-pai.

Not NULL

Define que um campo deve possuir um valor ou a instrução de inclusão ou alteração deve ser rejeitada.

Unique

Impõe que o valor resultante do campo ou conjunto de campos especificados seja único na tabela.

Check

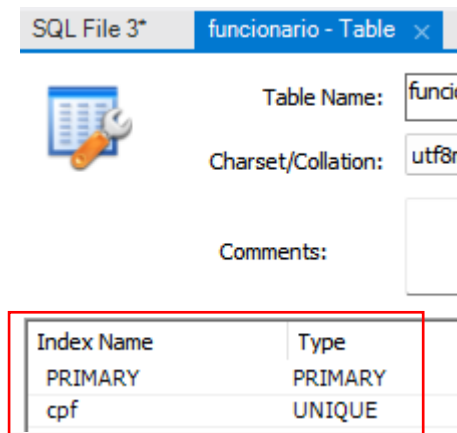
Permite codificar uma expressão que deve resultar verdadeiro (*true*) para a inclusão ou alteração do registro seja aceita.

Veremos seu uso logo mais.


Talvez você tenha percebido que *primary key* e *unique* foram utilizadas em nosso exemplo, diretamente especificadas na definição de um campo. Se não notou, por favor, reveja o exemplo.

Quando fazemos isso, por trás dos panos o SGBD define um nome automático para a *constraint* para nós.

Para nosso exemplo:



SQL File 3* funcionario - Table x

 Table Name:

Charset/Collation:

Comments:

Index Name	Type
PRIMARY	PRIMARY
cpf	UNIQUE

Agora, se uma *primary key* ou *unique* deve incluir vários campos, não podemos utilizar essa forma. Nesses casos, devemos especificar as *constraints* explicitamente.

Além de suprir a necessidade de especificar vários campos, esse método favorece documentação, uma vez que determinamos o nome que queremos para a restrição.

Vamos então refazer nossa instrução, agora cobrindo os pontos que haviam ficado pendentes:

```
create table if not exists treino.FUNCIONARIO (
    id            int            auto_increment,
    nome          varchar(50)    not null,
    sexo          char(1)        not null,
    nascto        date           not null,
    cid_natal     varchar(80)    default 'N/I',
    cpf           decimal(11,0)  not null unique,
    salario       decimal(7,2)   default 0.00,
    constraint FUNCIONARIO_PK primary key (id),
    constraint CHECK_SEXO check (sexo in ('M', 'F')),
    constraint CHECK_CPF check (length(cpf) = 11),
    constraint CHECK_SALARIO check (salario >= 0.00)
);
```

Bem, de fato não estamos verificando se o valor de CPF contém apenas dígitos. Esse tipo de validação requer programação SQL mais avançada, que cobriremos mais adiante no guia.

Para testar as restrições que definimos, executamos as seguintes instruções de inclusão de registro:

```
insert into treino.funcionario (nome, sexo, nascto,
cid_natal, cpf, salario)
values ('Joao', 'M', '1990/10/01', 'Santos',
'12345678901', 3500.00);
```

➔ Funcionará. O registro será criado.

```
Insert into treino.funcionario (nome, sexo, nascto,
cid_natal, cpf, salario)
values ('Maria', 'G', '1998/03/12', 'SCS', '12345678901',
7500.00);
```

➔ Rejeitará. Sexo inválido.

```
Insert into treino.funcionario (nome, sexo, nascto,
cid_natal, cpf, salario)
values ('Celso', 'm', '1998/05/06', null, '12345678901',
2000.00);
```

➔ Rejeitará. O CPF é o mesmo do primeiro registro.

```
Insert into treino.funcionario (nome, sexo, nascto,
cid_natal, cpf, salario)
values ('Carla', 'F', '2008/11/07', 'Curitiba',
'1234567890', 2200.00);
```

➔ Rejeitará. O CPF não contém 11 caracteres.

```
Insert into treino.funcionario (nome, sexo, nascto,
cid_natal, cpf, salario)
values ('Bia', 'F', '2006/09/03', 'Salvador',
'12345678901', -100.00);
```

➔ Rejeitará. Salário não pode ser negativo.

```
Insert into treino.funcionario (nome, sexo, nascto, cpf,
salario) values ('Jose', 'M', '2001/10/01',
'2222222222', 3500.00);
```

➔ Funcionará. cid_natal é um campo opcional, e como não especificamos nenhum valor, conterà **N/I** que é o valor default que definimos.

Ou seja, apenas dois registros foram aceitos:

	id	nome	sexo	nascto	cid_natal	cpf	salario
▶	1	Joao	M	1990-10-01	Santos	12345678901	3500.00
	2	Jose	M	2001-10-01	N/I	22222222222	3500.00

Não utilizaremos essa tabela mais. Ela serviu ao seu propósito. Vamos eliminá-la:

Execute:

```
drop table treino.funcionario;
```

DROP TABLE

A instrução é utilizada para excluir tabelas. Sua sintaxe é simples: **DROP TABLE <nome_tabela> [{CASCADE | RESTRICT}]**.

CASCADE ou RESTRICT são opções sobre como tratar tabelas-filhas. CASCADE orienta ao SGBD a tentar excluir todas as tabelas-filhas e seus dados; RESTRICT diz ao SGBD para parar se alguma tabela-filha existir. Caso não especificada, a opção padrão é RESTRICT.

Nem todo SGBD suporta CASCADE. MySQL, por exemplo, ignora o termo, sem gerar erro, e sempre executa com RESTRICT.

Logo, ao excluir tabelas relacionadas, o correto é irmos excluindo as tabelas-netas, depois as tabelas-filhas e finalmente as tabelas-mãe (a quantidade de níveis varia).

Existem comandos adicionais, para alterar a estrutura da tabela, criar e excluir outros tipos de estrutura. As veremos nos próximos capítulos.

O que vimos nesse capítulo

Conhecemos a instrução **CREATE TABLE** para criar tabelas.

Vimos com detalhes, **como especificar os campos** da tabela.

Aprendemos sobre **campos obrigatórios e opcionais**, bem como o uso de **valores default**.

Aprendemos sobre **restrições (*constraints*)**, e seu efeito prático na inclusão de dados.

Exercitamos o uso de **chaves primárias** e **chaves únicas**.

Utilizamos **check** para aplicar validações básicas em campos de dados.

Criamos nossa primeira tabela.

Realizamos algumas inclusões para **conferir o funcionamento das restrições**.

Vimos **como apagar uma tabela** utilizando **DROP TABLE**.

"Viva como se fosse morrer amanhã; Estude como se fosse viver para sempre"

Mahatma Ghandi.

Capítulo V – O modelo de dados a criar

A seguir serão especificadas as tabelas e relações que desejamos criar no esquema **treino**.

Será utilizada linguagem textual, como no capítulo anterior, para informar **o que** é desejado, **sem especificar como** implementar.

O desafio será interpretar as especificações e codificar as instruções SQL adequadas.

O modelo

Abrimos uma pequena empresa de venda de autopeças, e vamos montar um banco de dados para cadastrar os itens que comercializamos e as vendas realizadas.

Queremos controlar vendas diárias por **vendedor**, para que possamos premiar os melhores desempenhos. Dos vendedores, vamos cadastrar o nome e o número de crachá. O nome pode ter até 40 posições, e o crachá é numérico. Não deve ser possível cadastrar nomes ou crachás em duplicidade.

Precisamos de um cadastro de montadoras, veículos e seus componentes. Em geral as peças são para um modelo de veículo específico.

Primeiro, cadastraremos as diferentes **montadoras**. Apenas o nome é necessário. Por exemplo: FIAT, Ford, GM etc. Um campo com até 50 posições é suficiente. Só devemos tratar para que não seja possível alguém cadastrar em duplicidade.

Cada montadora produz diferentes modelos de veículos a cada ano. Então, é preciso cadastrar esses **modelos de veículos**.

Vamos armazenar a descrição do modelo num campo com até 80 posições, e o ano de fabricação em outro. Exemplos: "Onyx LTZ" ano 2020, "Voyage Comfortline", ano 2021.

Obviamente, o modelo de veículo deve estar corretamente relacionado com a montadora que o produz, e não deve ser permitido cadastrar o mesmo modelo em duplicidade.

Para cada diferente modelo de veículo cadastraremos vários **componentes**, por exemplo: "Velas de ignição", "Jogo de cabos de vela", "Junta do cabeçote", "Lâmpada dos faróis" etc.

Só vamos cadastrar a descrição inicialmente. E ela deve aceitar até 80 caracteres.

Não deve ser permitido cadastramento de componentes em duplicidade para um modelo de veículo.

Para cada componente que venderemos podem existir diversos fornecedores. Cada fornecedor define um código e um preço para o componente.

O primeiro passo será cadastrar os **fornecedores**, dos quais armazenaremos a razão social e o nome pelo qual é conhecido. Prever até 50 posições para cada campo é mais que suficiente.

Dos diferentes itens que cada fornecedor fabrica vamos registrar o código e o preço unitário. Para o código, 30 posições são suficientes. Já para o preço, devemos considerar duas casas decimais, e um valor máximo de R\$ 30.000,00.

Como cada item corresponde a um componente de um veículo, nesse cadastro armazenaremos a referência, e assim, teremos nosso **estoque**.

Nessa fase não vamos controlar o saldo. Isso não deve ser preocupação.

Devemos cuidar para que cada item de nosso estoque referencie um componente específico de um modelo de veículo e apenas um fornecedor.

Um fornecedor pode especificar mais de um código de item para o mesmo componente de um modelo de veículo. Isso não é um problema. Porém, considerando o componente específico, fornecedor e código do item para o fornecedor, não deve existir duplicidades.

Com os cadastros principais definidos, vamos gerar as tabelas para registro das vendas por vendedor.

Primeiro, vamos criar o cadastro de **vendas**. Devem ser registradas as seguintes informações: Data da venda; vendedor que realizou a venda; nome do cliente, cujo preenchimento será opcional, e 50 posições são suficientes; CPE, também opcional, com até 11 posições e o valor total, com duas decimais e máximo de R\$ 500.000,00.

O valor total não será informado. Numa fase posterior o sistema calculará e preencherá esse campo. Logo, ele não pode ser obrigatório.

Cada venda pode ser de um ou vários itens. Desses **itens vendidos** iremos registrar: o item de estoque vendido; a quantidade vendida e o preço unitário, com duas decimais.

Deve existir um campo de preço total, também com duas decimais.

De cada item o vendedor informará apenas o item de estoque e a quantidade. O preço unitário deverá ser obtido do estoque, e o preço total ficará em branco por agora.

Consistências desejadas:

1. Um vendedor só poderá ser excluído se não realizou nenhuma venda.
2. Um componente de um modelo de veículo só poderá excluído se ainda não houve nenhuma venda para ele.
3. Um veículo só poderá ser excluído se todos seus componentes puderem ser excluídos.
4. Uma montadora só poderá ser excluída se todos seus modelos de veículos puderem ser excluídos.
5. Um item de um fornecedor só poderá ser excluído se não houver vendas registradas para ele.
6. Um fornecedor só poderá ser excluído se todos seus itens puderem ser excluídos.
7. Uma venda só poderá ser excluída se ainda não existir nenhum item de venda relacionado cadastrado.

Já sabemos que faremos essas consultas sobre as vendas:

- a. Por data, vendedor e nome do cliente;
- b. Por nome do cliente;
- c. Por CPF do cliente.

Agora é com você!

Exemplo

Não seria justo deixar de dar uma ajuda aqui. Então, segue uma sugestão para a definição das tabelas de **montadoras** e **modelos de veículos**.

*“Primeiro, cadastraremos as diferentes **montadoras**. Apenas o nome é necessário. Por exemplo: FIAT, Ford, GM etc. Um campo com até 50 posições é suficiente. Só devemos tratar para que não seja possível alguém cadastrar em duplicidade.”*

```
CREATE TABLE IF NOT EXISTS montadora (  
  id INT NOT NULL AUTO_INCREMENT,  
  nome VARCHAR(50) NOT NULL,  
  PRIMARY KEY (id),  
  CONSTRAINT uk_nome UNIQUE (nome)  
);
```

*“Cada montadora produz diferentes modelos de veículos a cada ano. Então, é preciso cadastrar esses **modelos de veículos**.*

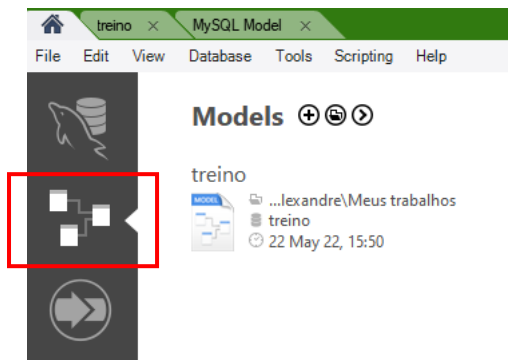
Vamos armazenar a descrição do modelo num campo com até 80 posições, e o ano de fabricação em outro. Exemplos: “Onyx LTZ” ano 2020, “Voyage Comfortline”, ano 2021.

Obviamente, o modelo de veículo deve estar corretamente relacionado com a montadora que o produz, e não deve ser permitido cadastrar o mesmo modelo em duplicidade.”

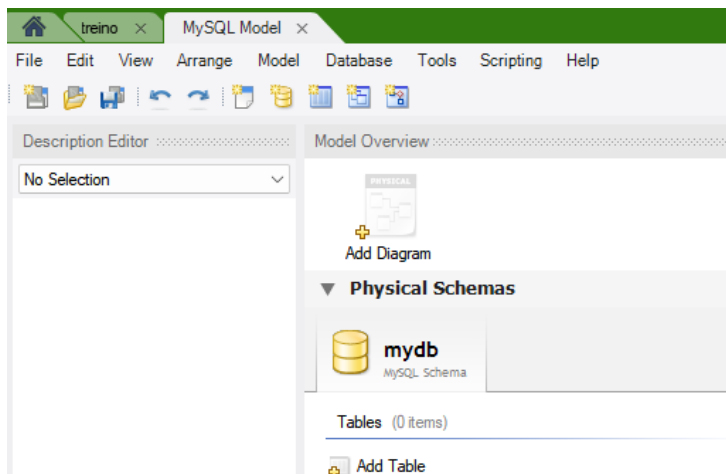
```
CREATE TABLE IF NOT EXISTS veiculo (  
  id INT NOT NULL AUTO_INCREMENT,  
  montadora_id INT NOT NULL,  
  modelo VARCHAR(80) NOT NULL,  
  ano_fabricacao SMALLINT NOT NULL,  
  PRIMARY KEY (id, modelo, ano_fabricacao),  
  CONSTRAINT fk_veiculo_montadora  
    FOREIGN KEY (montadora_id)  
      REFERENCES montadora (id)  
      ON DELETE CASCADE  
      ON UPDATE RESTRICT  
);
```

Dica

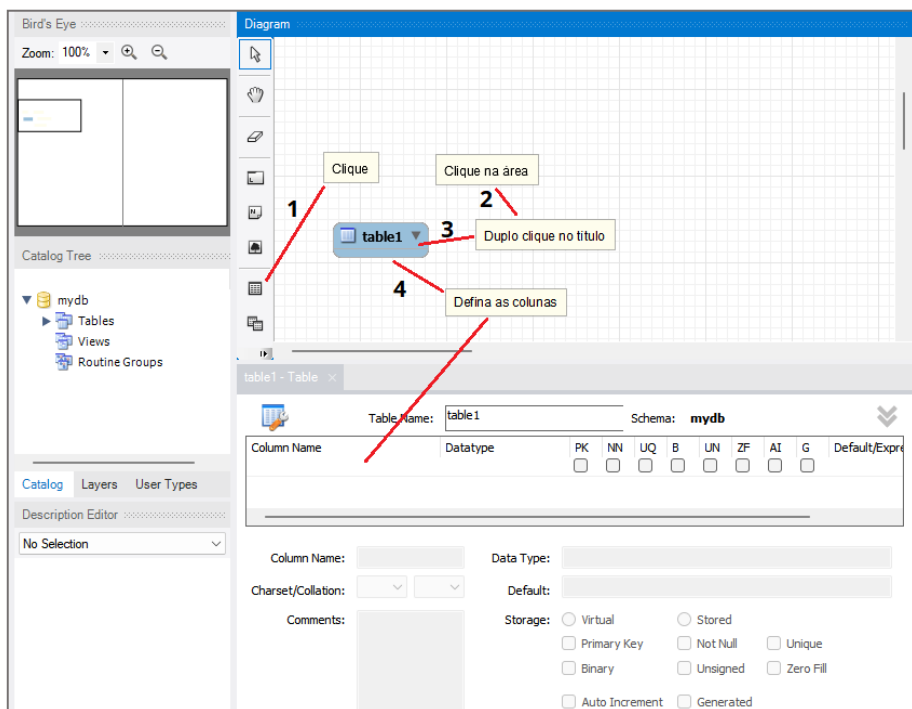
O Workbench possui uma interface para definição das tabelas de forma visual.



Ao clicar no ícone destacado, aparecerá a seguinte página:



Clique duplo em "Add Diagram" e comece a definir suas tabelas e relacionamentos.



Caso resolva usar essa ferramenta, só não esqueça de salvar seu modelo, de tempos em tempos e ao final.

Resumo das tabelas solicitadas:

Montadora
Veículo
Componente
Fornecedor
Estoque
Vendedor
Venda
Item de venda.

Boa sorte!

Capítulo VI – Solução para o modelo proposto

Se você tentou e conseguiu ou não criar o esquema, meus parabéns! Não conheço melhor forma de dominar uma tecnologia, que usá-la. Mas, se decidiu pular esse passo, tudo bem. Cada um tem uma forma de estudar e praticar.

Aqui está a minha solução para o modelo proposto:

```
-- Schema treino
```

```
USE treino;
```

```
-- Tabela montadora
```

```
CREATE TABLE IF NOT EXISTS montadora (  
  id INT NOT NULL AUTO_INCREMENT,  
  nome VARCHAR(50) NOT NULL,  
  PRIMARY KEY (id),  
  CONSTRAINT uk_nome UNIQUE (nome)  
);
```

```
-- Tabela veiculo
```

```
CREATE TABLE IF NOT EXISTS veiculo (  
  id INT NOT NULL AUTO_INCREMENT,  
  montadora_id INT NOT NULL,  
  modelo VARCHAR(80) NOT NULL,  
  ano_fabricacao SMALLINT NOT NULL,  
  PRIMARY KEY (id, modelo, ano_fabricacao),  
  CONSTRAINT fk_veiculo_montadora  
    FOREIGN KEY (montadora_id) REFERENCES montadora (id)  
    ON DELETE CASCADE ON UPDATE RESTRICT  
);
```

```

-----
-- Tabela componente
-----
CREATE TABLE IF NOT EXISTS componente (
    id            INT            NOT NULL AUTO_INCREMENT,
    veiculo_id    INT            NOT NULL,
    descricao     VARCHAR(80) NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT uk_descricao UNIQUE (veiculo_id, descricao),
    CONSTRAINT fk_componente_veiculo
        FOREIGN KEY (veiculo_id) REFERENCES veiculo (id)
        ON DELETE CASCADE ON UPDATE RESTRICT
);

```

```

-----
-- Tabela fornecedor
-----
CREATE TABLE IF NOT EXISTS fornecedor (
    id            INT            NOT NULL AUTO_INCREMENT,
    71nitá_social VARCHAR(50) NOT NULL,
    nome          VARCHAR(50) NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT uk_razao_social UNIQUE (71nitá_social),
    CONSTRAINT uk_nome UNIQUE (nome)
);

```

```

-----
-- Tabela estoque
-----
CREATE TABLE IF NOT EXISTS estoque (
    id                INT            NOT NULL AUTO_INCREMENT,
    componente_id     INT            NOT NULL,
    fornecedor_id     INT            NOT NULL,
    cod_comp_fornecedor VARCHAR(30) NOT NULL,
    preco             DECIMAL(7,2) NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT uk_cod_comp_fornecedor UNIQUE (cod_comp_fornecedor,
        fornecedor_id),
    CONSTRAINT fk_estoque_componente
        FOREIGN KEY (componente_id) REFERENCES componente (id)
        ON DELETE CASCADE ON UPDATE RESTRICT,
    CONSTRAINT fk_estoque_fornecedor
        FOREIGN KEY (fornecedor_id) REFERENCES fornecedor (id)
        ON DELETE CASCADE ON UPDATE RESTRICT
);

```

-- Tabela vendedor

```
CREATE TABLE IF NOT EXISTS vendedor (  
  id      INT          NOT NULL AUTO_INCREMENT,  
  nome    VARCHAR(40)  NOT NULL,  
  cracha  INT          NOT NULL,  
  PRIMARY KEY (id),  
  CONSTRAINT uk_vendedor_nome UNIQUE (nome),  
  CONSTRAINT uk_vendedor_cracha UNIQUE (cracha)  
);
```

-- Tabela venda

```
CREATE TABLE IF NOT EXISTS venda (  
  id          INT          NOT NULL AUTO_INCREMENT,  
  data_hora   DATETIME     NOT NULL,  
  vendedor_id INT          NOT NULL,  
  nome_cliente VARCHAR(50) NOT NULL,  
  cpf         VARCHAR(11)  DEFAULT 'NI',  
  valor_total DECIMAL(10,2),  
  PRIMARY KEY (id),  
  CONSTRAINT fk_venda_vendedor  
    FOREIGN KEY (vendedor_id) REFERENCES vendedor (id)  
    ON DELETE CASCADE ON UPDATE RESTRICT  
);
```

```
CREATE INDEX idx_nome_cliente ON venda (nome_cliente);
```

```
CREATE INDEX idx_cpf_cliente  ON venda (cpf);
```

```
CREATE INDEX idx_data_venda    ON venda (data_hora, vendedor_id,  
nome_cliente);
```

```
-- Tabela item_venda
```

```
CREATE TABLE IF NOT EXISTS item_venda (  
  id INT NOT NULL AUTO_INCREMENT,  
  venda_id INT NOT NULL,  
  estoque_id INT NOT NULL,  
  quantidade INT NOT NULL,  
  preco_unitario DECIMAL(7,2) NOT NULL,  
  preco_total DECIMAL(9,2),  
  PRIMARY KEY (id),  
  CONSTRAINT fk_item_venda_venda  
    FOREIGN KEY (venda_id) REFERENCES venda (id)  
    ON DELETE RESTRICT ON UPDATE RESTRICT,  
  CONSTRAINT fk_item_venda_estoque  
    FOREIGN KEY (estoque_id) REFERENCES estoque (id)  
    ON DELETE RESTRICT ON UPDATE RESTRICT  
);
```

👉 Linhas que iniciam com -- são comentários em SQL.
Para comentários de várias linhas, utilize /* e */

Explicando...

Tabelas “montadora” e “veículo”

As definições das tabelas **montadora** e **veículo** já foram explicadas no capítulo anterior (tópico Exemplo).

Tabela “componente”

*“Para cada diferente modelo de veículo cadastraremos vários **componentes**, por exemplo: “Velas de ignição”, “Jogo de cabos de vela”, “Junta do cabeçote”, “Lâmpada dos faróis” etc.*

Só vamos cadastrar a descrição inicialmente. E ela deve aceitar até 80 caracteres.

Não deve ser permitido cadastramento de componentes em duplicidade para um modelo de veículo.”

```
id INT NOT NULL AUTO_INCREMENT,  
veiculo_id INT NOT NULL,
```

```
descricao VARCHAR(80) NOT NULL,
```

Criamos o campo para a descrição, conforme especificado.

Adicionamos um campo ID que será a chave primária autoincrementada.

```
PRIMARY KEY (id),
```

A descrição do componente não pode repetir-se para um dado veículo (cadastramento em duplicidade):

```
CONSTRAINT uk_descricao UNIQUE (veiculo_id, descricao),
```

Cada componente deve relacionar-se com um modelo de veículo específico:

```
CONSTRAINT fk_componente_veiculo  
    FOREIGN KEY (veiculo_id) REFERENCES veiculo (id)  
    ON DELETE CASCADE ON UPDATE RESTRICT
```

Tabela “fornecedor”

“Para cada componente que venderemos podem existir diversos fornecedores. Cada fornecedor define um código e um preço para o componente.

*O primeiro passo será cadastrar os **fornecedores**, dos quais armazenaremos a razão social e o nome pelo qual é conhecido. Prever até 50 posições para cada campo é mais que suficiente.”*

```
CREATE TABLE IF NOT EXISTS fornecedor (  
    id INT NOT NULL AUTO_INCREMENT,  
    razao_social VARCHAR(50) NOT NULL,  
    nome VARCHAR(50) NOT NULL,  
    PRIMARY KEY (id),  
    CONSTRAINT uk_razao_social UNIQUE (razao_social),  
    CONSTRAINT uk_nome UNIQUE (nome)  
);
```

Além dos campos especificados, acrescentamos restrições que impedem a duplicidade de razão social e nome, individualmente.

E seguindo a linha de trabalho, um campo ID autoincrementado é a chave primária da tabela.

Tabela “estoque”

“Dos diferentes itens que cada fornecedor fabrica vamos registrar o código e o preço unitário. Para o código, 30 posições são suficientes. Já para o preço, devemos considerar duas casas decimais, e um valor máximo de R\$ 30.000,00.

*Como cada item corresponde a um componente de um veículo, nesse cadastro armazenaremos a referência, e assim, teremos nosso **estoque**.*

Nessa fase não vamos controlar o saldo. Isso não deve ser preocupação.

Devemos cuidar para que cada item de nosso estoque referencie um componente específico de um modelo de veículo e apenas um fornecedor.

Um fornecedor pode especificar mais de um código de item para o mesmo componente de um modelo de veículo. Isso não é um problema. Porém, considerando o componente específico, fornecedor e código do item para o fornecedor, não deve existir duplicidades.”

```
CREATE TABLE IF NOT EXISTS estoque (  
    id INT NOT NULL AUTO_INCREMENT,  
    componente_id INT NOT NULL,  
    fornecedor_id INT NOT NULL,  
    cod_comp_fornecedor VARCHAR(30) NOT NULL,  
    preco DECIMAL(7,2) NOT NULL,  
    PRIMARY KEY (id),  
    CONSTRAINT uk_cod_comp_fornecedor UNIQUE (cod_comp_fornecedor,  
    fornecedor_id),  
    CONSTRAINT fk_estoque_componente  
        FOREIGN KEY (componente_id) REFERENCES componente (id)  
        ON DELETE CASCADE ON UPDATE RESTRICT,  
    CONSTRAINT fk_estoque_fornecedor  
        FOREIGN KEY (fornecedor_id) REFERENCES fornecedor (id)  
        ON DELETE CASCADE ON UPDATE RESTRICT  
);
```

`id` `INT` `NOT NULL AUTO_INCREMENT,`
Chave primária da tabela

`componente_id` `INT` `NOT NULL,`
Referência para a tabela “componente”

`fornecedor_id` `INT` `NOT NULL,`
Referência para a tabela “fornecedor”

`cod_comp_fornecedor` `VARCHAR(30)` `NOT NULL,`
Código do componente para o fornecedor

`preco` `DECIMAL(7,2)` `NOT NULL,`
Preço do componente, segundo esse fornecedor

`CONSTRAINT uk_cod_comp_fornecedor UNIQUE (cod_comp_fornecedor, fornecedor_id),`

O código do componente não pode repetir-se para um mesmo fornecedor, embora diferentes fornecedores possam usar o mesmo código (reflita sobre isso).

O código é o primeiro campo, pois assim já temos um índice geral de códigos, que poderia ser utilizado para agilizar uma eventual tela de pesquisa pelos vendedores em um sistema que a empresa viesse a criar.

`CONSTRAINT fk_estoque_componente`
`FOREIGN KEY (componente_id) REFERENCES componente (id)`
`ON DELETE CASCADE ON UPDATE RESTRICT,`

Relaciona a tabela **estoque** com a tabela **componente**.

`CONSTRAINT fk_estoque_fornecedor`
`FOREIGN KEY (fornecedor_id) REFERENCES fornecedor (id)`
`ON DELETE CASCADE ON UPDATE RESTRICT`

Relaciona a tabela **estoque** com a tabela **fornecedor**.

Tabela “vendedor”

“Queremos controlar vendas diárias por **vendedor**, para que possamos premiar os melhores desempenhos. Dos vendedores, vamos cadastrar o nome e o número de crachá. O nome pode ter até 40 posições, e o crachá é numérico. Não deve ser possível cadastrar nomes ou crachás em duplicidade.”

```
CREATE TABLE IF NOT EXISTS vendedor (  
  id      INT          NOT NULL AUTO_INCREMENT,  
  nome    VARCHAR(40) NOT NULL,  
  cracha  INT          NOT NULL,  
  PRIMARY KEY (id),  
  CONSTRAINT uk_vendedor_nome UNIQUE (nome),  
  CONSTRAINT uk_vendedor_cracha UNIQUE (cracha)  
);
```

Acredito que os campos **id**, **nome** e **cracha** já sejam autoexplicativos para você a essa altura; assim como a cláusula *primary key* da instrução.

Repare então, como as duas restrições *UNIQUE* implementam o controle contra duplicidades.

Tabela “venda”

“Primeiro, vamos criar o cadastro de **vendas**. Devem ser registradas as seguintes informações: Data da venda; vendedor que realizou a venda; nome do cliente, cujo preenchimento será opcional, e 50 posições são suficientes; CPE, também opcional, com até 11 posições e o valor total, com duas decimais e máximo de R\$ 500.000,00.

O valor total não será informado. Numa fase posterior o sistema calculará e preencherá esse campo. Logo, ele não pode ser obrigatório.”

```
CREATE TABLE IF NOT EXISTS venda (  
  id          INT          NOT NULL AUTO_INCREMENT,  
  data_hora   DATETIME     NOT NULL,  
  vendedor_id INT          NOT NULL,
```

```

nome_cliente VARCHAR(50),
cpf          VARCHAR(11)  DEFAULT 'NI',
valor_total  DECIMAL(10,2) DEFAULT 0,
PRIMARY KEY (id),
CONSTRAINT fk_venda_vendedor
    FOREIGN KEY (vendedor_id) REFERENCES vendedor (id)
    ON DELETE CASCADE ON UPDATE RESTRICT
);

```

Os campos requisitados estão presentes.

O relacionamento da venda com o vendedor está definido, com o campo **vendedor_id** e a chave estrangeira (*foreign key*).

Note que o campo **nome_cliente** aceita NULL. Como especificado, é um campo de preenchimento opcional.

Para o campo **cpf**, embora não especificado, adotamos um padrão: preencher com NI caso não informado.

Criamos o campo **valor_total**, opcional, como especificado no enunciado. Instruções de inclusão podem omitir esse campo.

Tabela “item_venda”

*“Cada venda pode ser de um ou vários itens. Desses **itens vendidos** iremos registrar: o item de estoque vendido; a quantidade vendida e o preço unitário, com duas decimais.*

Deve existir um campo de preço total, também com duas decimais.

De cada item o vendedor informará apenas o item de estoque e a quantidade. O preço unitário deverá ser obtido do estoque, e o preço total ficará em branco por agora.”

```

CREATE TABLE IF NOT EXISTS item_venda (
    id            INT            NOT NULL AUTO_INCREMENT,
    venda_id      INT            NOT NULL,
    estoque_id    INT            NOT NULL,
    quantidade    INT            NOT NULL,
    preco_unitario DECIMAL(7,2) NOT NULL,

```

```

preco_total    DECIMAL(9,2),
PRIMARY KEY (id),
CONSTRAINT fk_item_venda_venda
    FOREIGN KEY (venda_id) REFERENCES venda (id)
    ON DELETE RESTRICT ON UPDATE RESTRICT,
CONSTRAINT fk_item_venda_estoque
    FOREIGN KEY (estoque_id) REFERENCES estoque (id)
    ON DELETE RESTRICT ON UPDATE RESTRICT
);

```

Temos o campo ID autoincrementado, os campos de relacionamento com as tabelas-pai (venda_id e estoque_id), o campo para quantidade e os campos de preço unitário e total.

O preco total não será informado nas inclusões, logo, é opcional na definição (não temos “NOT NULL” na definição do campo).



Note a vantagem de uso de chaves autoincrementadas para todas as tabelas: um **item de venda** refere-se a um registro da tabela **estoque**, que por sua vez se relaciona com um **componente**. Componente se relaciona com **veículo**, que se relaciona com **montadora**. Apenas um campo do tipo INT em cada nível é suficiente para construir todas essas relações, sem ficarmos repetindo códigos das tabelas-mãe nas tabelas-filhas.



Repare na cláusula ON DELETE das duas *foreign keys* dessa tabela. Todas as demais chaves estrangeiras empregam “CASCADE”, enquanto aqui, empregamos “RESTRICT”.

Lembra-se do item “Consistências desejadas” da especificação?

“Consistências desejadas:

1. Um vendedor só poderá ser excluído **se não realizou nenhuma venda**.
2. Um componente de um modelo de veículo só poderá excluído se ainda **não houve nenhuma venda** para ele.
3. Um veículo só poderá ser excluído se todos seus componentes puderem ser excluídos.
4. Uma montadora só poderá ser excluída se todos seus modelos de veículos puderem ser excluídos.
5. Um item de um fornecedor só poderá ser **excluído se não houver vendas registradas para ele**.
6. Um fornecedor só poderá ser excluído se todos seus itens puderem ser excluídos.
7. Uma venda só poderá ser excluída se ainda **não existir nenhum item de venda relacionado** cadastrado.”

Todas essas condições são automaticamente tratadas pela forma como as chaves estrangeiras foram especificadas.

Otimização das consultas

Finalmente:

Já sabemos que faremos essas consultas sobre as vendas:

- d. Por data, vendedor e nome do cliente;
- e. Por nome do cliente;
- f. Por CPF do cliente.

Suprimos assim:

```
CREATE INDEX idx_data_venda    ON venda (data_hora, vendedor_id,  
nome_cliente);
```

```
CREATE INDEX idx_nome_cliente ON venda (nome_cliente);
```

```
CREATE INDEX idx_cpf_cliente  ON venda (cpf);
```

O que vimos nesse capítulo

Uma **solução para o esquema proposto** foi apresentada e explicada.

Aprendemos como **chaves estrangeiras servem para atender regras de negócio** relacionadas com alterações e exclusões, além de proporcionar consistência. Em bases muito complexas, garantir consistência via código de aplicação pode ser muito difícil.

Vimos também, **como índices são criados com o intuito de otimizar consultas**.

"Ele não sabia que era impossível. Foi lá e fez."
Jean Cocteau

Capítulo VII – Mais comandos DDL

Veremos aqui as instruções mais utilizados na manutenção de objetos de um esquema, deixando de lado, propositalmente, aqueles relacionados com objetos que ainda não vimos.

CREATE [UNIQUE] INDEX

No final do capítulo anterior vimos como adicionar índices para uma tabela. A sintaxe é:

```
CREATE [UNIQUE] INDEX nome_indice ON (col1 [{ASCending | DESCending}] [, col2 [{ASCending | DESCending}]...);
```

UNIQUE é opcional. Quando especificado o índice não aceitará inclusões ou alterações que resultem em duplicidades, considerando as colunas especificadas para o índice.

Para cada coluna podemos especificar se a ordem é crescente (ASCENDING ou ASC) ou decrescente (DESCENDING ou DESC). Caso não especificado, o padrão é crescente (ASCENDING).

O nome do índice deve ser único no esquema.

Exemplos:

```
CREATE UNIQUE INDEX idx_cidade ON pais (UF, cidade);
```

Um índice de cidades do Brasil que não aceita duplicidades para cada UF.

```
CREATE INDEX idx_venda_mes ON venda (mes_ano DESC, valor DESC);
```

Um índice de vendas que ordena por mês decrescentemente (os mais recentes primeiro) e dos valores maiores para os menores a cada mês.

DROP INDEX

Utilizada para excluir índices.

Sintaxe: **DROP INDEX nome_indice;**

ALTER TABLE

Utilizamos ALTER TABLE para realizar uma série de manutenções na estrutura das tabelas de nossos bancos de dados.

ADD COLUMN

Executamos **ALTER TABLE ADD COLUMN** para adicionar colunas a uma tabela existente.

Sintaxe:

```
ALTER TABLE nome_tabela  
ADD COLUMN nome_col defs_col;
```

Sendo que *defs_col* representa os termos para definição da coluna, como visto em CREATE TABLE.

DROP COLUMN

Utilizada para excluir colunas da tabela.

Sintaxe:

```
ALTER TABLE nome_tabela  
DROP COLUMN nome_col;
```

ADD CONSTRAINT

Permite adicionar restrições para uma tabela, tais como PRIMARY KEY, FOREIGN KEY, UNIQUE e CHECK.

Sintaxe:

```
ALTER TABLE nome_tabela  
ADD CONSTRAINT nome_constraint defs_constraint;
```

Onde *defs_constraint* representa os termos para definição da restrição, como visto nos capítulos anteriores.

DROP CONSTRAINT

Exclui uma *constraint* de uma tabela.

Sintaxe:

```
ALTER TABLE nome_tabela  
DROP CONSTRAINT nome_constraint;
```

Existem mais opções de uso, mesmo para essas instruções; e claro, há mais instruções relacionadas com outros tipos de objeto. Até mesmo os analistas mais experientes recorrem aos manuais, quando necessário. Então, não se preocupe em decorar todos eles, ou conhecer tudo de uma vez, ok?

O que vimos nesse capítulo

Vimos como realizar manutenção em tabelas e índices do um banco de dados:

- Criar índices;
- Excluir índices;
- Adicionar colunas numa tabela;
- Excluir colunas de uma tabela;
- Adicionar *constraints*;
- Excluir *constraints*;

“Existem três formas de fazer algo: A minha forma; A sua forma;
A forma empregada pelos outros...”

Eu

Capítulo VIII – Testando o esquema

Após criarmos ou modificarmos um esquema, é preciso testar.

Fazemos isso incluindo, alterando e excluindo registros, buscando validar as restrições, informando valores consistentes e inconsistentes.

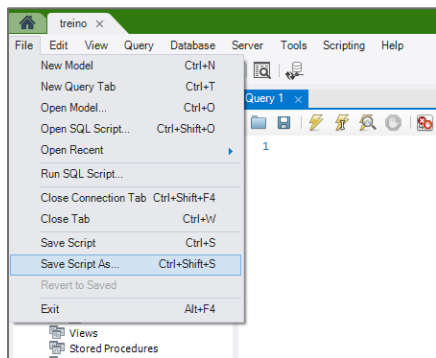
Isso pode ser feito interativamente (comando a comando) ou em lote, através de scripts.

Scripts SQL

Um script SQL nada mais é que um arquivo-texto contendo instruções a serem executadas em sequência pelo SGBD.

De fato, é uma boa prática construir scripts, pois eles podem ser reutilizados em futuras manutenções, evitando um bocado de digitação.

No Workbench podemos facilmente digitar uma série de instruções e então salvar como um script, através da opção **"File" → Save Script As...**



Então, futuramente carregamos o script para execução com a opção **File → Open SQL Script...** e salvamos horas de trabalho.

Scripts também podem ser enviados para outros executarem, e de modo geral, é assim que bancos de dados são planejados, testados e depois colocados em produção. Criamos e testamos num servidor de desenvolvimento e publicamos nos servidores de produção.

Os desenvolvedores, em geral, só possuem acesso aos servidores de desenvolvimento e enviam os scripts para a equipe que tem acesso aos servidores de produção.

No capítulo anterior listamos as instruções para a criação das tabelas e índices. De fato, elas compõem um script.

Você pode realizar o download desse script através desse link:



https://1drv.ms/u/s!AmhDenEg1Sjyhp5vXoZg4gn_GrOPZw?e=GVDyAi

DDL x DML

Essa é uma boa hora para adicionar mais um conceito ao nosso conhecimento sobre a linguagem SQL. As instruções dessa linguagem são organizadas em dois grupos:

DDL – Data Definition Language

É o conjunto de instruções utilizadas para definição ou alteração das estruturas de um banco de dados relacional. CREATE DATABASE; CREATE TABLE; CREATE INDEX; ALTER TABLE; DROP TABLE e outros, são exemplos desse conjunto.

DML – Data Manipulation Language

Refere-se ao conjunto de instruções para manipulação dos dados. INSERT, UPDATE e DELETE são alguns exemplos.

Você pode consultar o guia oficial dos bancos a seguir através dos links.

MySQL:



<https://dev.mysql.com/doc/refman/8.0/en/language-structure.html>

Postgres:



<https://www.postgresql.org/docs/current/sql-commands.html>

Oracle:



<https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/>

Microsoft SQL Server:



<https://docs.microsoft.com/pt-br/sql/t-sql/language-reference?view=sql-server-ver15>

Portanto, para testar seu esquema você utilizará instruções DML.

Para realizar consultas utilizará a instrução SELECT.

Para incluir registros, INSERT.

Para alterar dados, UPDATE.

Para excluir registros, DELETE.

Essas são, certamente, as quatro instruções SQL mais comuns.



INSERT, UPDATE e DELETE operam apenas em uma tabela. Já SELECT, que é muito poderosa, pode buscar dados de várias tabelas numa única instrução.

Vamos começar inserindo dados.

INSERT

Para inserir dados “manualmente” a sintaxe é:

```
INSERT [INTO] [esquema].tabela [(col1 [,col2 [,col3  
...]])] VALUES (val1 [,val2 [,val3 ...]]);
```

Exemplos:

```
INSERT INTO montadora (nome) VALUES ("FIAT");  
INSERT INTO fornecedor (razao_social, nome)  
VALUES ("Bosch S/A", "Bosch");  
INSERT INTO montadora VALUES (0, "Renault");
```

Para cada campo especificado um valor deve ser informado na cláusula VALUES.

Se nenhum campo for especificado, deve-se fornecer um valor para cada coluna da tabela, na ordem em que foram definidos; inclusive campos autoincrementados e opcionais. Repare o último exemplo acima.

Não há problema em utilizar mais de uma linha para codificar a instrução. Só não faça isso para um valor *string* (entre aspas).

A instrução se encerra no ponto-e-vírgula.

- Para campos CHAR e VARCHAR informe o valor entre aspas simples ou aspas duplas;
- Para campos numéricos não usamos aspas;
- Para valores com casas decimais, use ponto (.) para separar;
- Datas são informadas entre aspas também. A forma ("dd-mm/aaaa"; "aaaa-mm-dd"; "dd/mm/aaaa"; "aaaa/mm/dd"...) depende do servidor.

No MySQL você pode executar o seguinte comando para descobrir o formato esperado: `select now();`

E o que acontece se forneço dados para um campo auto incrementado?



Resposta: Depende. Alguns SGBD podem simplesmente ignorar o valor informado, armazenando o próximo valor gerado internamente. Outros podem aceitar, verificando se não existe ainda na tabela.

Note como o MySQL trata:

```
SELECT * FROM montadora ORDER BY id;
```

	id	nome
►	1	FIAT
	2	Ford
	3	GM/Chevrolet
	4	Volkswagen/VW

Próximo ID = 5

```
INSERT INTO montadora VALUES (6,  
"Renault");
```

	id	nome
►	1	FIAT
	2	Ford
	3	GM/Chevrolet
	4	Volkswagen/VW
	6	Renault

Acatou 6, pulando o 5.

```
INSERT INTO montadora VALUES (5, "Honda ");
```

	id	nome
►	1	FIAT
	2	Ford
	3	GM/Chevrolet
	4	Volkswagen/VW
	5	Honda
	6	Renault

Acatou 5, que ainda não existia.

INSERT INTO montadora VALUES (0, "Toyota");

	id	nome
▶	1	FIAT
	2	Ford
	3	GM/Chevrolet
	4	Volkswagen/VW
	5	Honda
	6	Renault
	7	Toyota

Ignorou o 0 (zero) e atribuiu 7, apesar do último valor inserido ter sido 5, porque identificou que 6 já existia.

Agora, se tentamos "forçar" o valor do campo:

INSERT INTO montadora VALUES (3, "Peugeot");

❌ 32 13:45:15 INSERT INTO montadora VALUES (3, "Peugeot") Error Code: 1062. Duplicate entry '3' for key 'montadora.PRIMARY'



Sempre que possível, evite fornecer valores para campos autoincrementados. Deixe que o SGBD cuide deles.

Para inserir dados "copiados" de uma ou mais tabelas:

Podemos inserir registros numa tabela a partir de qualquer consulta, desde que as colunas correspondam em tipo e tamanho. Isso pode ser muito útil.

A sintaxe básica é:

INSERT [INTO] [esquema].tabela [(col1 [,col2 [,col3 ...]])] SELECT ... ;

Imagine que tivéssemos uma tabela denominada **funcionario** em nosso banco de dados, contendo as colunas nome e crachá que utilizamos na tabela vendedores, e uma coluna cargo, cujo código 123 representasse "vendedor".

Após criarmos nossa tabela vendedor, poderíamos preenchê-la com uma única instrução INSERT, da seguinte forma:

```
INSERT INTO vendedor (nome, cracha)
SELECT nome, cracha FROM funcionario WHERE cargo = 123;
Usando variáveis com INSERT
```

Se você inserir as montadoras e em seguida desejar cadastrar os modelos de veículos deparará com uma situação interessante.

Quando cadastramos as montadoras não definimos o valor para o campo ID, já que ele é autoincrementado. Então, como saberemos o ID de cada uma? Teremos que consultar e anotar para inserir os veículos?

Embora isso seja um caminho, há uma forma melhor. Em scripts SQL podemos armazenar o resultado de consultas em variáveis, desde que o retorno seja apenas um registro.

Então, podemos utilizar o valor da variável nas próximas instruções; por exemplo, em INSERTs.

Exemplo:

```
INSERT INTO montadora (nome) values ("FIAT");
SELECT id INTO @vIdMontadora FROM montadora WHERE nome =
"FIAT";
INSERT INTO veiculo (montadora_id, modelo,
ano_fabricacao) values (@vIdMontadora, "Uno", 2020);
```

Inseridos a montadora "FIAT". Em seguida, usamos SELECT para armazenar o id na variável @vIdMontadora; então, inserimos o veículo "Uno" utilizando nossa variável para o campo montadora_id.

Podemos também, inserir vários registros numa única instrução INSERT. Algo semelhante às tuplas da linguagem Python.

Exemplo:

```
insert into vendedor (nome, cracha)
values ('Fabio', 400)
      , ('Carlos', 401)
      , ('Francisca', 402)
      , ('Helio', 403);
```

Há várias possibilidades adicionais no uso de INSERT. Sinta-se à vontade para explorar. Para nossos propósitos aqui, vimos o suficiente.

UPDATE

A instrução UPDATE é utilizada para modificar campos (colunas) de um registro.



“Colunas” e “campos” e são sinônimos quando estamos nos referindo a registros de um banco de dados relacional.

Podemos atualizar apenas um registro, vários ou todos.

A sintaxe básica é:

```
UPDATE tabela SET campor1=valor1 [, campo2=valor2 [,...]]
[WHERE condição];
```

Só é possível atualizar dados de uma tabela por vez.

Todas as *constraints* que foram definidas são verificadas, e se não forem respeitadas, o comando é cancelado.

Vamos ver um exemplo.

Imagine que, sem querer, você executou o seguinte comando:

```
INSERT INTO montadora (nome) values ("FAIT");
```

Puxa vida, o nome da montadora foi digitado errado.

Para corrigir, executaria o seguinte:

```
UPDATE montadora SET nome="FIAT" WHERE nome="FAIT";
```



Muito cuidado com o "UPDATE da morte !!!"

Um erro mais comum do que muitos imaginam, é a execução de instruções UPDATE sem a cláusula WHERE.

De fato, WHERE é uma cláusula opcional, e quando não especificada, faz com que o SGBD atualize (ou tente atualizar) todos os registros de uma tabela.

Imagine o que ocorreria num sistema de controle de estoques, a execução da seguinte instrução:

```
UPDATE estoque SET codigo="000", descricao="Parafuso";
```

DELETE

Para excluir um ou mais registros de uma tabela utilizamos a instrução DELETE. A sintaxe é:

```
DELETE FROM tabela [WHERE condição];
```

Exemplo:

```
DELETE FROM venda WHERE data_hora < "2021-01-01";
```

Isso excluiria todas as vendas e itens relacionados anteriores a Janeiro de 2021.



Muito cuidado com o "DELETE da morte !!!"

Executar um comando DELETE sem a cláusula WHERE é possível, e em alguns casos, motivo de ataques cardíacos, comas e outros problemas mais.

Por exemplo:

DELETE FROM venda;

Simplesmente excluirá todas as vendas de nosso banco de dados.

START TRANSACTION, ROLLBACK e COMMIT

É possível que você tenha ficado preocupado com o UPDATE ou DELETE da morte. Não me surpreenderia.

Felizmente, os inventores dos bancos de dados relacionais previram esse perigo, e criaram proteções contra esses potenciais desastres: o **conceito de transação**.

Transação

Uma transação é uma unidade de inclusões, alterações e/ou exclusões, em uma ou várias tabelas, que só é visível para quem a iniciou, até que seja **confirmada** ou **cancelada**.

O uso é simples: O usuário ou programa inicia uma transação, executa os vários comandos de atualização e ao final, confirma ou cancela as operações.

A instrução **START TRANSACTION;** inicia uma transação.

COMMIT; confirma as alterações.

ROLLBACK; desfaz as alterações.

Durante uma transação podemos executar instruções **SELECT** para avaliar os resultados para decidir se vamos ou não confirmar.

Vamos praticar?

Acompanhe o exemplo a seguir.

```
select * from item_venda;
```

id	venda_id	estoque_id	quantidade	preco_unitario	preco_total
1	1	1	1	600.00	600.00
2	2	44	1	440.00	440.00
3	2	92	1	240.00	240.00
4	2	164	1	103.00	103.00
5	2	188	1	104.00	104.00
6	3	96	1	400.00	400.00
7	4	112	1	76.00	76.00
8	4	136	1	76.00	76.00
9	5	116	1	102.00	102.00
10	5	140	1	102.00	102.00
11	6	23	1	750.00	750.00
12	7	29	1	660.00	660.00
13	7	77	1	360.00	360.00
14	8	67	1	220.00	220.00
15	9	97	1	80.00	80.00
16	9	121	1	78.00	78.00
17	10	181	1	70.00	70.00
18	11	42	1	590.00	590.00
19	12	11	3	940.00	2820.00
20	12	93	2	280.00	560.00
21	13	62	1	290.00	290.00
22	14	102	1	150.00	150.00
23	14	126	1	150.00	150.00
24	15	152	1	99.00	99.00
25	15	176	1	94.00	94.00

```
start transaction;  
delete from item_venda;  
select * from item_venda;
```

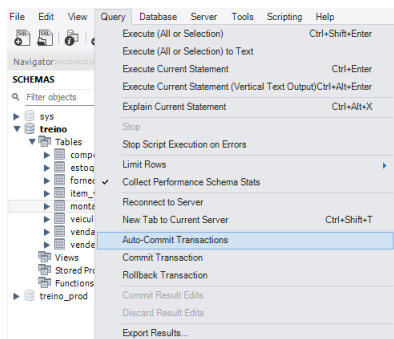
id	venda_id	estoque_id	quantidade	preco_unitario	preco_total
NULL	NULL	NULL	NULL	NULL	NULL

```
rollback;  
select * from item_venda;
```

id	venda_id	estoque_id	quantidade	preco_unitario	preco_total
1	1	1	1	600.00	600.00
2	2	44	1	440.00	440.00
3	2	92	1	240.00	240.00
4	2	164	1	103.00	103.00
5	2	188	1	104.00	104.00
6	3	96	1	400.00	400.00
7	4	112	1	76.00	76.00
8	4	136	1	76.00	76.00
9	5	116	1	102.00	102.00
10	5	140	1	102.00	102.00
11	6	23	1	750.00	750.00
12	7	29	1	660.00	660.00
13	7	77	1	360.00	360.00
14	8	67	1	220.00	220.00
15	9	97	1	80.00	80.00
16	9	121	1	78.00	78.00
17	10	181	1	70.00	70.00
18	11	42	1	590.00	590.00
19	12	11	3	940.00	2820.00
20	12	93	2	280.00	560.00
21	13	62	1	290.00	290.00
22	14	102	1	150.00	150.00
23	14	126	1	150.00	150.00
24	15	152	1	99.00	99.00
25	15	176	1	94.00	94.00

Os dados retornaram exatamente que estavam antes do DELETE.

É comum que ferramentas visuais como o Workbench estejam configuradas para confirmar transações automaticamente.



Recomendo que sempre desative, pois isso deixa o ambiente mais seguro. A praticidade aqui, definitivamente não compensa.

Transações e campos autoincrementados

Devido questões de concorrência, o próximo valor para um campo autoincrementado não retorna ao valor inicial se uma transação for cancelada por rollback.

Transações e comandos DDL



Instruções DDL não respeitam transações. Portanto, se uma instrução DROP TABLE (apagar uma tabela) for executada, ROLLBACK não a trará de volta.

Exemplo:

#	Time	Action	Message
✓ 1	18:18:37	create table temp (c1 int, c2 int)	0 row(s) affected
✓ 2	18:18:44	start transaction	0 row(s) affected
✓ 3	18:18:44	drop table temp	0 row(s) affected
✓ 4	18:18:46	rollback	0 row(s) affected
✗ 5	18:18:46	select * from temp LIMIT 0, 1000	Error Code: 1146. Table 'temp' doesn't exist

Transações e concorrência

Durante uma transação o SGBD “trava” os registros inseridos, alterados ou excluídos, para prevenir que outras conexões tentem alterar em paralelo. Esse mecanismo é conhecido como **LOCK** (trava).

Enquanto usando um banco local, e sendo o único usuário, tudo bem. Mas em ambientes compartilhados, e principalmente em produção, isso é um problema. Pode-se parar sistemas inteiros

com uma transação parada, aguardando pelo *commit* ou *rollback*.



Em ambientes compartilhados seja rápido. Execute, salve o resultado dos SELECT para avaliar com calma, e execute o ROLLBACK para liberar o banco para os demais usuários.

Perdeu o medo dos “UPDATE e DELETE da morte”, né?



SELECT – Parte I

Você já viu a instrução SELECT por aqui. Utilizamos essa instrução para consultar dados de uma tabela.

De fato, SELECT é a instrução mais popular da linguagem SQL, e uma das mais complexas, tamanha quantidade de variações que possui.

Nesse estágio de trabalho, utilizaremos SELECT apenas para validar os dados que forem inseridos nas tabelas. A sintaxe básica é:

```
SELECT {*|col1[,col2[,...]]}  
FROM tabela  
WHERE condição  
ORDER BY {expressão ou lista};
```

Quando queremos todas as colunas utilizamos um asterisco, e se queremos apenas algumas colunas, utilizamos uma lista separada por vírgula.

Se desejarmos filtrar apenas alguns registros, utilizamos uma expressão lógica através da cláusula WHERE. E se queremos que

os resultados estejam ordenados por um ou mais colunas específicas, empregamos ORDER BY.

Na cláusula ORDER BY podemos especificar os nomes das colunas desejadas ou seu índice de posição, iniciando em 1.

Exemplos:

```
SELECT * FROM fornecedor ORDER BY RAZAO_SOCIAL;
```

- ➔ Todas as colunas da tabela **fornecedor**, com registros ordenados pela coluna **razao_social** em ordem crescente.

```
SELECT nome, cracha  
FROM vendedor  
WHERE nome LIKE "J%"  
ORDER BY cracha DESC;
```

- ➔ Nome e crachá dos vendedores cujo nome inicia pela letra J, ordenados por crachá, ordem decrescente.

Bem, isso é suficiente para que você possa popular seu esquema com alguns dados, consultar o conteúdo das tabelas, realizar alguns testes de alteração e exclusão e validar as restrições.

Leve o tempo que julgar necessário nessas atividades, principalmente se está iniciando com a linguagem SQL.

Mas antes de encerrar...

Deixo nesse link, um script que preenche o esquema apresentado aqui com dados de teste. Se você criou as tabelas exatamente como apresentado, poderá simplesmente abri-lo no Workbench e executá-lo para alimentar as tabelas.



<https://1drv.ms/u/s!AmhDenEg1Sjyhp5leekPJhGwhkp73w?e=hQoWwy>

O que vimos nesse capítulo

Vimos como **scripts SQL** são úteis, e como os criamos.

Conhecemos os conceitos **DDL** e **DML**.

Reunimos links para a **documentação oficial da linguagem SQL** dos SGBD mais populares.

Conhecemos **INSERT**, **UPDATE** e **DELETE**.

Aprendemos sobre como MySQL trata campos **autoincrementados quando valores são fornecidos** em instruções **INSERT**.

Aprendemos como **"copiar" dados de uma tabela para outra**.

Fomos apresentados aos **"UPDATE e DELETE da morte"**.

Aprendemos a **prevenir desastres** utilizando **START TRANSACTION**, **ROLLBACK** e **COMMIT**.

Aprendemos sobre campos autoincrementados em transações e **a importância de agilidade em transações** em virtude da concorrência.

Tivemos uma **introdução sobre SELECT**, suficiente para as consultas iniciais.

"Na vida nós devemos ter raízes, não âncoras. Raíz alimenta, âncora imobiliza."

Mário Sérgio Cortella

Capítulo IX – Indo além das consultas simples

Minha intenção é que chegue aqui com o esquema criado e tabelas preenchidas com dados de teste.

Que tenha conseguido realizar algumas consultas em cada tabela com SELECT, e eventualmente treinado o uso de INSERT, UPDATE e DELETE.

Idealmente, fez uso de START TRANSACTION e testou ROLLBACK e COMMIT para compreender seu funcionamento.

Essa trilha, a meu ver, facilita a compreensão dos exemplos desse capítulo, porque o(a) amigo(a) leitor(a) estará familiarizado com as tabelas e seus conteúdos e relacionamentos.

SELECT – parte II

Até aqui usamos SELECT para consultar dados de tabelas individuais. Nesse cenário:

- 👉 Indicamos quais colunas queremos, ou todas, usando ***** (**asterisco**);
- 👉 Opcionalmente filtramos os registros desejados, adicionando a cláusula **WHERE**.
- 👉 Opcionalmente indicamos a ordem em que queremos os dados, através da cláusula **ORDER BY**.

Mas podemos fazer muito mais:

- ✓ Podemos usar funções sobre os dados das colunas;
- ✓ “Unir” dados de várias tabelas;
- ✓ Realizar operações sobre dados agrupados, tais como obter somas, médias e contagens.

E há mais.

Uso de funções e expressões

Vamos iniciar pelo uso de funções. SQL define algumas funções comuns para o tratamento dos dados das colunas. Os SGBD mais populares permitem ainda, que você defina funções personalizadas.



Esse último ponto não abordaremos aqui, mas julguei importante mencionar.

Considere a seguinte consulta e seu resultado (com os dados de teste do link fornecido):

```
1 • select * from vendedor;
```

Result Grid		 Filter Rows:	
	id	nome	cracha
▶	6	Joao da Silva	200
	7	Jose Oliveira	201
	8	Maria Francisca	202
	9	Pedro Paulo	203
	10	Paula Souza	204
*	NULL	NULL	NULL

Agora, vamos utilizar algumas funções e demonstrar os resultados:

```
select upper(nome) as nome, cracha from vendedor;
```

nome	cracha
JOAO DA SILVA	200
JOSE OLIVEIRA	201
MARIA FRANCISCA	202
PEDRO PAULO	203
PAULA SOUZA	204

→ `upper()` converte todas as letras em maiúsculas.



Note esse detalhe: `upper(nome)` **as nome**

Podemos “renomear” qualquer coluna que quisermos numa consulta através do termo **as**. Isso não afeta o nome da coluna, apenas o título que ela tem nos resultados.

É muito útil quando utilizamos funções e expressões, como nesse exemplo.

```
select lower(nome) as nome_min, cracha from vendedor;
```

	nome_min	cracha
▶	joao da silva	200
	jose oliveira	201
	maria francisca	202
	pedro paulo	203
	paula souza	204

→ `lower()` converte todas as letras em minúsculas.

```
select substr(nome, 1, 5), cracha from vendedor;
```

	substr(nome, 1, 5)	cracha
▶	Joao	200
	Jose	201
	Maria	202
	Pedro	203
	Paula	204

→ `substr()` retornar partes de um campo char ou varchar. O valor é a posição inicial e o segundo a quantidade de caracteres desejados.



Quando não utilizamos `as` o título da coluna é a própria expressão empregada.

```
select max(cracha) from vendedor;
```

	max(cracha)
▶	204

→ **substr()** retorna o maior valor de uma coluna.

```
select max(nome) from vendedor;
```

	max(nome)
▶	Pedro Paulo

Em dados alfanuméricos isso representa o último valor, considerando a ordem alfabética.

Similar a **max()** temos **min()**, **avg()**, **std()**, **count()** e outras mais.

min() retorna o menor valor;

avg() retorna a média dos valores numéricos;

std() retorna o desvio-padrão de valores numéricos;

count() contabiliza quantos valores retornaram.

Fique à vontade para experimentá-las.



MUITO CUIDADO!

Note o exemplo a seguir:

```
select nome, max(cracha) as cracha from vendedor;
```

	nome	cracha
▶	Joao da Silva	204

Isso está correto? Bem, está considerando o que pediu:
“Traga o nome e o maior crachá da tabela vendedor”

Mas reveja a tabela completa:



```
1 • select * from vendedor;
```

	id	nome	cracha
▶	6	Joao da Silva	200
	7	Jose Oliveira	201
	8	Maria Francisca	202
	9	Pedro Paulo	203
	10	Paula Souza	204
•	NULL	NULL	NULL

João da Silva possui o crachá 200, não 204!

O SGBD interpretou que você desejava o valor da coluna "nome" do primeiro registro e o maior valor da coluna "cracha", logo devolveu os valores apropriados.

Não é um erro, mas para obter o vendedor com o maior crachá deveríamos executar essa instrução:

```
select nome, cracha from vendedor where cracha =  
(select max(cracha) from vendedor);
```

A quantidade de funções é expressiva em qualquer SGBD; explore o quanto desejar.

Vejamos agora, como aplicar expressões nos valores das colunas retornadas.

```
select nome + cracha from vendedor;
```

	nome + cracha
▶	200
	201
	202
	203
	204

Ops!!! Não funcionou como previsto?

Em muitas linguagens de programação empregamos o sinal de adição (+) para concatenar valores. No SQL não.

Apesar de alguns SGBD disponibilizarem uma alternativa, recomendo que utilize a função **concat()**, pois isso facilita eventuais migrações de servidor.

`select concat(nome, " ", cracha) from vendedor;`

	concat(nome, " ", cracha)
▶	Joao da Silva 200
	Jose Oliveira 201
	Maria Francisca 202
	Pedro Paulo 203
	Paula Souza 204

Mas, se desejamos realmente realizar contas com o valor de uma coluna, sem problemas!

`select (cracha * 1000 / 2) + 5 as cracha_conta from vendedor;`

	cracha_conta
▶	100005.0000
	100505.0000
	101005.0000
	101505.0000
	102005.0000

E como podemos ter valores não-inteiros, o SGBD converteu a coluna apropriadamente automaticamente.

CASE WHEN

Agora, vejamos a poderosa expressão **case-when-then-else**:

```
select
case
when nome like "J%" then upper(nome)
else lower(nome)
end as nome_j_destacados,
cracha
from vendedor;
```

	nome_j_destacados	cracha
▶	JOAO DA SILVA	200
	JOSE OLIVEIRA	201
	maria francisca	202
	pedro paulo	203
	paula souza	204

Com essa expressão aplicamos verdadeiros IF-ELSEIF...ELSE nos valores de uma coluna.

LIKE



Note que empreguei **like** para comparar os nomes. **Like** permite comparar CHAR e VARCHAR com os caracteres coringa % e _.

% significa "qualquer expressão" e _ representa qualquer caractere (uma posição).

Nesse exemplo, like "J%" pode ser lido como "Todo nome que inicia com J".

Agora veja esse exemplo:

```
select
case when nome like "_a%" then upper(nome)
else lower(nome)
end as nome_2a
from vendedor;
```

	nome_2a
▶	joao da silva
	jose oliveira
	MARIA FRANCISCA
	PAULA SOUZA
	pedro paulo

Aqui, solicitamos aplicar upper() a todo nome cujo segundo caractere fosse "a" e lower() para os demais nomes.

SELECT ... GROUP BY

A instrução SELECT conta com a cláusula **GROUP BY** para permitir o uso de dados agrupados.

Para exercitamos essa cláusula vamos usar a tabela **venda**.

```
select * from venda;
```

	id	data_hora	vendedor_id	nome_cliente	cpf	valor_total
▶	1	2022-04-01 00:00:00	1	Celso	HULL	HULL
	2	2022-04-01 00:00:00	2		HULL	HULL
	3	2022-04-01 00:00:00	3		HULL	HULL
	4	2022-04-01 00:00:00	1		HULL	HULL
	5	2022-04-01 00:00:00	1		HULL	HULL
	6	2022-04-02 00:00:00	2	Sr. Gilson	HULL	HULL
	7	2022-04-02 00:00:00	4		HULL	HULL
	8	2022-04-02 00:00:00	3		HULL	HULL
	9	2022-04-02 00:00:00	3	Aline	1100023009	HULL
	10	2022-04-02 00:00:00	2		HULL	HULL
	11	2022-04-02 00:00:00	2		HULL	HULL

Vamos contar pedidos por data:

```
select data_hora, count(*)  
from venda  
group by data_hora;
```

	data_hora	count(*)
▶	2022-04-01 00:00:00	5
	2022-04-02 00:00:00	5
	2022-04-03 00:00:00	5
	2022-04-04 00:00:00	5
	2022-04-05 00:00:00	5

Agora, vamos contabilizar por data e por vendedor:

```
select data_hora, vendedor_id, count(*)  
from venda  
group by data_hora, vendedor_id;
```

	data_hora	vendedor_id	count(*)
▶	2022-04-01 00:00:00	1	3
	2022-04-01 00:00:00	2	1
	2022-04-01 00:00:00	3	1
	2022-04-02 00:00:00	2	2
	2022-04-02 00:00:00	3	2
	2022-04-02 00:00:00	4	1
	2022-04-03 00:00:00	1	1
	2022-04-03 00:00:00	2	2
	2022-04-03 00:00:00	3	1
	2022-04-03 00:00:00	4	1
	2022-04-04 00:00:00	2	3
	2022-04-04 00:00:00	4	1
	2022-04-04 00:00:00	5	1
	2022-04-05 00:00:00	1	2
	2022-04-05 00:00:00	2	1
	2022-04-05 00:00:00	3	2



Como regra, é mandatório colocar na cláusula GROUP BY todas as colunas para as quais não está sendo aplicada uma função de sumarização, tais como max(), min(), sum(), count() etc. Não fazer isso é um erro.

Veja:

```
select data_hora, vendedor_id, count(*)  
from venda  
group by data_hora;
```

	data_hora	vendedor_id	count(*)
▶	2022-04-01 00:00:00	1	5
	2022-04-02 00:00:00	2	5
	2022-04-03 00:00:00	1	5
	2022-04-04 00:00:00	2	5
	2022-04-05 00:00:00	1	5

A falta de **vendedor_id** fez o SGBD omitir dados.

Agora, digamos que queremos, desse resultado, apenas os vendedores que venderam mais de um pedido em cada dia.

```
select data_hora, vendedor_id, count(*)  
from venda  
where count(*) > 1  
group by data_hora, vendedor_id;
```

Isso parece coerente, mas será que funciona?

Error Code: 1111. Invalid use of group function

Não, não funciona.

Não é correto tentarmos utilizar count(*) no WHERE, porque ele é avaliado antes do GROUP BY. Assim, nesse momento o SGBD ainda não sabe o valor dessa contagem.

Como resolvemos?

```
select data_hora, vendedor_id, count(*)  
from venda  
group by data_hora, vendedor_id  
having count(*) > 1;
```


Resposta: utilizando **HAVING**

HAVING

A cláusula HAVING é a versão da cláusula WHERE para os dados agrupados. Ou seja, condições que queremos aplicar aos registros individuais especificamos na cláusula WHERE, e condições que queremos aplicar aos dados agrupados, especificamos na cláusula HAVING.

Desafio:

Considerando a tabela **estoque**, executa uma instrução SELECT...GROUP BY que contabilize os produtos com preço acima de 400,00 de cada fornecedor.

Qual fornecedor tem mais produtos nessa condição?

	fornecedor_id	count(*)
▶	1	28
	2	26

Resposta: Fornecedor 1.

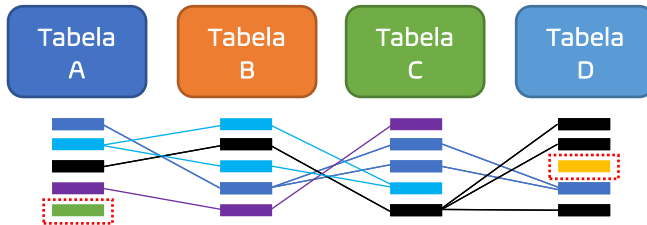
Agora vamos ver diferentes formas de consolidar colunas e registros de diferentes tabelas.

JOIN

O processo de juntar dados de duas ou mais tabelas é denominado JOIN (junção).

Acho produtivo compreender visualmente, como o SGBD faz as junções: elas ocorrem sempre “da esquerda para a direita”:

```
SELECT *
FROM TabelaA
JOIN TabelaB ON...
JOIN TabelaC ON...
JOIN TabelaD ON...
```



Os pequenos retângulos na figura anterior representam registros.

Primeiro, a tabela A é percorrida, e para cada registro dessa tabela são identificados os correspondentes na tabela B, que pode resultar em nenhum, um ou muitos; para esse conjunto, então, são percorridos os registros da tabela C, que são adicionados; finalmente, para cada registro do conjunto A-B-C, são pesquisados os registros da tabela D.

O padrão de cores foi utilizado para exemplificar as relações hipoteticamente identificadas.

As relações completas, ou seja, que localizaram registros nas quatro tabelas foram destacados.

E há registros sem qualquer relação: o último registro da tabela A (verde) e o terceiro registro da tabela D (amarelo). O que ocorre com esses registros será uma opção nossa, como veremos mais adiante.

O padrão é excluir do resultado os registros que não possuírem correspondências em todas as tabelas.

Nessa configuração nosso resultado seria:



Ou seja, os registros da tabela A que têm correspondentes nas tabelas B, C e D.

Existe também um processo de junção por **combinação**. Nesse processo cada registro da tabela da esquerda é relacionado com todos os registros da tabela da direita, e assim sucessivamente.

A sintaxe é:

```
SELECT ... FROM tab1, tab2, ..., tabN WHERE ... ORDER BY ...;
```

Exemplo:

```
select *  
from vendedor, montadora, veiculo  
where vendedor.nome <> "Pedro Paulo"  
and montadora.nome <> "Ford";
```

id	nome	cracha	id	nome	id	montadora_id	modelo	ano_fabricacao
5	Paula Souza	204	1	FIAT	1	1	Uno	2020
5	Paula Souza	204	3	GM/Chevrolet	1	1	Uno	2020
5	Paula Souza	204	4	Volkswagen/VW	1	1	Uno	2020
3	Maria Francisca	202	1	FIAT	1	1	Uno	2020
3	Maria Francisca	202	3	GM/Chevrolet	1	1	Uno	2020
3	Maria Francisca	202	4	Volkswagen/VW	1	1	Uno	2020
2	Jose Oliveira	201	1	FIAT	1	1	Uno	2020
2	Jose Oliveira	201	3	GM/Chevrolet	1	1	Uno	2020
2	Jose Oliveira	201	4	Volkswagen/VW	1	1	Uno	2020
1	Joao da Silva	200	1	FIAT	1	1	Uno	2020
1	Joao da Silva	200	3	GM/Chevrolet	1	1	Uno	2020

Utilizei uma consulta sem coerência para demonstrar o conceito apenas. Note que podemos utilizar WHERE para limitar os registros a considerar em cada tabela.

Essa consulta resultou em 288 registros, logo, acima está apenas a parte inicial.

Note que para “Paula Souza” foram relacionadas todas as montadoras, e para cada par “Paula Souza” x montadora, o SGBD relacionou todos os veículos (no trecho só aparece o primeiro).

Tipos de JOIN

Lembra-se que afirmei que “o que ocorre com registros sem relações em outras tabelas” é uma opção nossa?

Temos dois tipos de JOIN: **INNER** e **OUTER**.

Considerando as tabelas A e B, num INNER JOIN, são relacionados os registros da tabela A que possuem um ou mais correspondentes na tabela B.

Já o OUTER JOIN trará todos os registros da tabela A, independente de existirem registros relacionados na tabela B.

Exemplos:

Inseri três montadoras na base de teste, para ter certeza que nenhum veículo está associado ainda.

```
insert into montadora (nome)
values (“Renault”), (“Peugeot”), (“Honda”);
commit;
```

Lembre-se: Temos FIAT, Ford, GM e VW com veículos cadastrados.

INNER JOIN

Na query a seguir, solicitamos ao SGBD que exclua “FIAT” e “FORD”.

```
select *
from montadora
inner join veiculo on veiculo.montadora_id = montadora.id
where montadora.nome <> "FIAT"
    and montadora.nome <> "Ford"
order by montadora.nome;
```

Ela é equivalente à seguinte query:

```
select *
from montadora
join veiculo on veiculo.montadora_id = montadora.id
where montadora.nome <> "FIAT"
    and montadora.nome <> "Ford"
order by montadora.nome;
```

Ou seja, não é preciso codificar "inner", pois esse é o padrão.

montadora inner veiculo é sinônimo de **montadora join veiculo**

id	nome	id	montadora_id	modelo	ano_fabricacao
3	GM/Chevrolet	13	3	Celta	2020
3	GM/Chevrolet	14	3	Celta	2021
3	GM/Chevrolet	15	3	Celta	2022
3	GM/Chevrolet	16	3	Onyx	2020
3	GM/Chevrolet	17	3	Onyx	2021
3	GM/Chevrolet	18	3	Onyx	2022
4	Volkswagen/VW	19	4	Gol	2020
4	Volkswagen/VW	20	4	Gol	2021
4	Volkswagen/VW	21	4	Gol	2022
4	Volkswagen/VW	22	4	Polo	2020
4	Volkswagen/VW	23	4	Polo	2021
4	Volkswagen/VW	24	4	Polo	2022

Resultado:

Como esperado, nenhum registro das novas montadoras, afinal não há veículos relacionados a elas ainda.

Agora vamos executar a mesma consulta com **outer join**:

OUTER JOIN

```
select *  
from montadora  
left outer join veiculo on veiculo.montadora_id =  
montadora.id  
where montadora.nome <> "FIAT"  
and montadora.nome <> "Ford"  
order by montadora.nome;
```

id	nome	id	montadora_id	modelo	ano_fabricacao
3	GM/Chevrolet	13	3	Celta	2020
3	GM/Chevrolet	14	3	Celta	2021
3	GM/Chevrolet	15	3	Celta	2022
3	GM/Chevrolet	16	3	Onyx	2020
3	GM/Chevrolet	17	3	Onyx	2021
3	GM/Chevrolet	18	3	Onyx	2022
7	Honda	NULL	NULL	NULL	NULL
6	Peugeot	NULL	NULL	NULL	NULL
5	Renault	NULL	NULL	NULL	NULL
4	Volkswagen/VW	19	4	Gol	2020
4	Volkswagen/VW	20	4	Gol	2021
4	Volkswagen/VW	21	4	Gol	2022
4	Volkswagen/VW	22	4	Polo	2020
4	Volkswagen/VW	23	4	Polo	2021
4	Volkswagen/VW	24	4	Polo	2022

Note que as novas montadoras apareceram. E como não há registros na tabela de veículos, todas as colunas oriundas daquela tabela estão preenchidas com o valor especial **null**.

RIGHT (OUTER) JOIN

Finalmente, cito a variante **RIGHT JOIN**, menos comumente utilizada. Quando desejamos que os registros do lado direito

sem correspondente na tabela da esquerda façam parte do resultado, essa é a sintaxe empregada.

Exemplo:

```
start transaction;
insert into vendedor (nome, cracha) values ("AAA",500);
insert into vendedor (nome, cracha) values ("BBB",501);
insert into vendedor (nome, cracha) values ("CCC",502);
```

```
select * from venda
right join vendedor on venda.vendedor_id = vendedor.id;
```

Note: incluímos três vendedores para os quais certamente não

id	data_hora	vendedor_id	nome_cliente	cpf	valor_total	id	nome	cracha
19	2022-04-04 00:00:00	2		NULL	NULL	2	Jose Oliveira	201
20	2022-04-04 00:00:00	2		NULL	NULL	2	Jose Oliveira	201
22	2022-04-05 00:00:00	2	Ofic. Helio	NULL	NULL	2	Jose Oliveira	201
3	2022-04-01 00:00:00	3		NULL	NULL	3	Maria Franci...	202
8	2022-04-02 00:00:00	3		NULL	NULL	3	Maria Franci...	202
9	2022-04-02 00:00:00	3	Aline	110...	NULL	3	Maria Franci...	202
14	2022-04-03 00:00:00	3		NULL	NULL	3	Maria Franci...	202
23	2022-04-05 00:00:00	3		NULL	NULL	3	Maria Franci...	202
25	2022-04-05 00:00:00	3		NULL	NULL	3	Maria Franci...	202
7	2022-04-02 00:00:00	4		NULL	NULL	4	Pedro Paulo	203
12	2022-04-03 00:00:00	4	Ofic. Helio	NULL	NULL	4	Pedro Paulo	203
16	2022-04-04 00:00:00	4		NULL	NULL	4	Pedro Paulo	203
17	2022-04-04 00:00:00	5	Ofic. Helio	NULL	NULL	5	Paula Souza	204
NULL	NULL	NULL	NULL	NULL	NULL	8	AAA	500
NULL	NULL	NULL	NULL	NULL	NULL	9	BBB	501
NULL	NULL	NULL	NULL	NULL	NULL	10	CCC	502

há

vendas. Na query acima, **vendas** é a tabela da **esquerda** e **vendedor** a tabela da **direita**.

Os novos vendedores aparecem, mesmo sem qualquer venda.

Aliás de nome de tabela

Ficar digitando o nome completo da tabela é tedioso. Sobre tudo se a query for complexa, com muitas colunas e relações. Assim como utilizamos o termo **AS** para renomear colunas, podemos

definir um aliás mais sucinto para os nomes de tabelas, utilizando também o termo **AS**.

Exemplo:

```
SELECT
    VD.id as id_venda
  , VD.data_hora
  , concat(V.modelo, " ", V.ano_fabricacao) as veiculo
  , I.quantidade as qtd
  , C.descricao as componente
  , I.preco_unitario
  , I.quantidade * I.preco_unitario as preco_total
  , F.nome fornecedor
  , E.cod_comp_fornecedor as cod_forn
FROM venda as VD
JOIN item_venda as I ON I.venda_id = VD.id
JOIN estoque    as E ON E.id = I.estoque_id
JOIN componente as C ON C.id = E.componente_id
JOIN veiculo    as V ON V.id = C.veiculo_id
JOIN montadora  as M ON M.id = V.montadora_id
JOIN fornecedor as F ON F.id = E.fornecedor_id
ORDER BY VD.id, C.descricao;
```

id_venda	data_hora	veiculo	qtd	componente	preco_unitario	preco_total	fornecedor	cod_forn
1	2022-04-01 00:00:00	Uno 2020	1	Velas de ignicao	600.00	600.00	Bosch	VL.BSH.UNO.2020
2	2022-04-01 00:00:00	Gol 2021	1	Jogo de cabos de vela	240.00	240.00	NGK	JC.NGK.GOL.2021
2	2022-04-01 00:00:00	Gol 2021	1	Palheta lado motorista	103.00	103.00	Dyna	LE.DYN.GOL.2021
2	2022-04-01 00:00:00	Gol 2021	1	Palheta lado passageiro	104.00	104.00	Dyna	LD.DYN.GOL.2021
2	2022-04-01 00:00:00	Gol 2021	1	Velas de ignicao	440.00	440.00	NGK	VL.NGK.GOL.2021
3	2022-04-01 00:00:00	Polo 2022	1	Jogo de cabos de vela	400.00	400.00	NGK	JC.NGK.POL.2022
4	2022-04-01 00:00:00	Onyx 2020	1	Palheta lado motorista	76.00	76.00	Bosch	PLMOT.BSH.ONX.2020
4	2022-04-01 00:00:00	Onyx 2020	1	Palheta lado passageiro	76.00	76.00	Bosch	PLPAS.BSH.ONX.2020

Vamos praticar?

O que vimos nesse capítulo?

Exploramos a instrução **SELECT** com funções e expressões.

Vimos alguns usos de **min()**, **max()**, **avg()**, **std()** e **count()**.

Mostramos como utilizar **concat()** para concatenar valores de várias colunas.

Vimos que é possível realizar cálculos com colunas numéricas.

Aprendemos a utilizar **CASE-WHEN-THEN-ELSE-END**.

Exercitamos o uso de **LIKE** em colunas **CHAR** e **VARCHAR**.

Conhecemos a cláusula **GROUP BY** para agrupar dados.

Demonstramos o uso de **HAVING** para aplicar condições nos dados agrupados.

Aprendemos a consolidar dados de diferentes tabelas com o uso de **JOIN**.

Conhecemos a junção de tabelas por combinação.

Vimos a diferença entre **INNER JOIN** e **OUTER JOIN**.

Mostramos como usar **RIGHT JOIN**.

Descobrimos que podemos usar **AS** para nomes de tabelas, diminuindo a quantidade de digitação necessária.

Persistência é o caminho para o sucesso.

Capítulo X – Mais sobre SELECT

Limit ou Top

Notou que sempre obtemos todos os dados quando executamos nossas consultas? Em tabelas muito grandes isso pode não ser conveniente.

Podemos limitar a quantidade de dados empregando LIMIT ou TOP (diferentes fornecedores suportam diferentes sintaxes).

MS SQL Server, por exemplo, suporta **SELECT TOP nnn ... FROM**

Já MySQL suporta **SELECT ... FROM ... LIMIT nnn;**

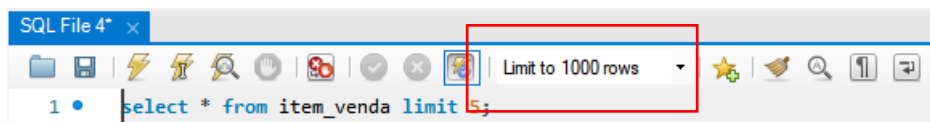
Onde nnn é a quantidade máxima de registros que queremos.

Exemplo:

```
select * from item_venda limit 5;
```

id	venda_id	estoque_id	quantidade	preco_unitario	preco_total
1	1	1	1	600.00	NULL
2	2	44	1	440.00	NULL
3	2	92	1	240.00	NULL
4	2	164	1	103.00	NULL
5	2	188	1	104.00	NULL

Adicionalmente, repare que o Workbench usa uma configuração padrão, mesmo que você não especifique nada, para prevenir problemas. No topo da área de consulta é exibido o limite implícito empregado:



Distinct

Uma necessidade comum nas consultas, é a remoção de duplicidades.

Suponha que nos solicitaram uma lista dos diferentes componentes vendidos por vendedor, não importando o modelo de veículo. São desejados apenas o nome do vendedor e a descrição do componente.

Inicialmente, executamos a seguinte consulta:

```
SELECT V.nome, C.descricao
FROM venda      VD
JOIN item_venda I ON I.venda_id = VD.id
JOIN estoque    E ON E.id = I.estoque_id
JOIN componente C ON C.id = E.componente_id
JOIN vendedor   V ON V.id = VD.vendedor_id;
ORDER BY 1, 2;
```



Note que não empreguei o AS para os nomes de tabela nas cláusulas FROM e JOIN. Simplesmente adicionei o apelido para a tabela, e isso foi aceito. Fica a dica!

O resultado (apenas registros iniciais) seria:

nome	descricao
Joao da Silva	Jogo de cabos de vela
Joao da Silva	Palheta lado motorista
Joao da Silva	Palheta lado motorista
Joao da Silva	Palheta lado motorista
Joao da Silva	Palheta lado passageiro
Joao da Silva	Palheta lado passageiro
Joao da Silva	Palheta lado passageiro
Joao da Silva	Velas de ignicao
Joao da Silva	Velas de ignicao
Jose Oliveira	Jogo de cabos de vela
Jose Oliveira	Jogo de cabos de vela
Jose Oliveira	Jogo de cabos de vela
Jose Oliveira	Palheta lado motorista
Jose Oliveira	Palheta lado motorista
Jose Oliveira	Palheta lado motorista

Temos duplicidades, porque utilizamos várias tabelas relacionadas para chegar ao resultado, mas não queremos as demais colunas.

DISTINCT existe para solucionar isso. Como o termo indica, somente as linhas não duplicadas são retornadas pela instrução **SELECT**.

Repetindo nossa consulta:

```
SELECT DISTINCT V.nome, C.descricao  
FROM venda      VD  
JOIN item_venda I ON I.venda_id = VD.id  
JOIN estoque    E ON E.id = I.estoque_id  
JOIN componente C ON C.id = E.componente_id  
JOIN vendedor   V ON V.id = VD.vendedor_id;  
ORDER BY 1, 2;
```

nome	descricao
Joao da Silva	Jogo de cabos de vela
Joao da Silva	Palheta lado motorista
Joao da Silva	Palheta lado passageiro
Joao da Silva	Velas de ignicao
Jose Oliveira	Jogo de cabos de vela
Jose Oliveira	Palheta lado motorista
Jose Oliveira	Palheta lado passageiro
Jose Oliveira	Velas de ignicao
Maria Francisca	Jogo de cabos de vela
Maria Francisca	Palheta lado motorista
Maria Francisca	Palheta lado passageiro
Paula Souza	Jogo de cabos de vela
Paula Souza	Velas de ignicao
Pedro Paulo	Jogo de cabos de vela
Pedro Paulo	Velas de ignicao

Subqueries

A instrução SELECT é realmente poderosa. Creio que o capítulo anterior, em particular, demonstra isso. Mas ainda não vimos tudo. Outro recurso muito útil é o uso de subqueries. De fato, já até já o utilizamos anteriormente.

Uma subquery é uma query empregada no contexto de outra query, ou seja, um SELECT dentro de outro SELECT.

Exemplo:

```
SELECT nome, cracha
FROM vendedor
WHERE cracha = (SELECT max(cracha) FROM vendedor);
```

Essa consulta retornará o nome e o crachá do vendedor que possui o maior número de crachá da tabela.

O primeiro passo realizado pelo SGBD é a query que calcula o maior número de crachá. O valor de retorno então, é empregado na cláusula WHERE para filtrar os registros da tabela vendedor que correspondam à comparação.

Outro exemplo:

```
select *
from (select F.id, F.nome, E.cod_comp_fornecedor,
E.componente_id as comp, E.preco
      from fornecedor as F join estoque E on
E.fornecedor_id = F.id) as F
join componente as C on C.id = F.comp
join veiculo as V on V.id = C.veiculo_id;
```

Note que a subquery agora está na cláusula FROM, atuando como uma nova tabela de nome F.

Certamente não vimos tudo sobre a instrução `SELECT`, mas creio que para o intuito desse guia, temos o suficiente.

O domínio e descoberta dos detalhes adicionais vem da prática e necessidades que surgirem.

O que vimos nesse capítulo

Descobrimos como limitar a quantidade de dados retornada em nossas consultas com **LIMIT** ou **TOP**.

Aprendemos a finalidade da cláusula **DISTINCT**.

Conhecemos as **subqueries**, e como ampliam ainda mais o poder da instrução **SELECT**.

“Você não pode controlar a vida, mas pode controlar como reage ao que ela lhe proporciona.”
Eu e um monte de filósofos.

Capítulo XI – VIEWS

Digitar instruções SQL é até divertido. Principalmente quando estamos aprendendo e praticando. Com tempo construímos instruções cada vez mais sofisticadas.

No uso profissional, os bancos de dados crescem, em quantidade de dados e número de tabelas criadas. Isso aumenta a necessidade por novas consultas.

Uma VIEW é uma instrução SELECT salva, que gera uma pseudo-tabela. Assim, não precisamos ficar lembrando o código da query, facilitando a vida de quem desenvolve e de quem usa a consulta.

Criar uma VIEW é muito simples. Prefixamos nossa instrução SELECT com CREATE (OR REPLACE) VIEW, como a seguir:

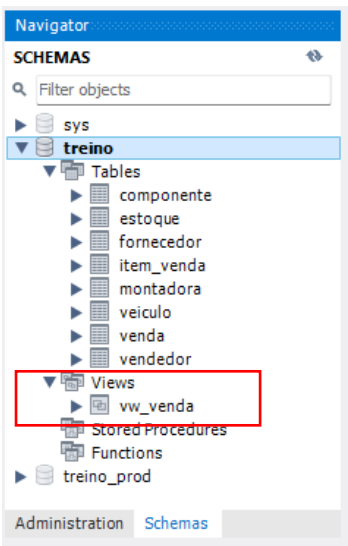
```
CREATE OR REPLACE VIEW vw_venda AS  
SELECT  
    VD.id as id_venda  
    , VD.data_hora  
    , concat(V.modelo, " ", V.ano_fabricacao) as veiculo  
    , I.quantidade as qtd  
    , C.descricao as componente  
    , I.preco_unitario  
    , I.quantidade * I.preco_unitario as preco_total  
    , F.nome fornecedor  
    , E.cod_comp_fornecedor as cod_forn  
FROM venda as VD  
JOIN item_venda as I ON I.venda_id = VD.id  
JOIN estoque    as E ON E.id = I.estoque_id  
JOIN componente as C ON C.id = E.componente_id  
JOIN veiculo    as V ON V.id = C.veiculo_id  
JOIN montadora  as M ON M.id = V.montadora_id  
JOIN fornecedor as F ON F.id = E.fornecedor_id  
ORDER BY VD.id, C.descricao;
```


Ao executar essa instrução nenhum dado retornará.

Se nenhuma mensagem de erro for apresentada, a nova VIEW foi criada.

No MySQL Workbench podemos confirmar isso atualizando a visão do esquema (clitando com o botão direito do mouse na área “Schemas”) e clicando na opção Refresh All na base do menu que surge.

Agora, expanda o item Views, e a nova visão deve aparecer lá:



Agora, ao invés de digitar toda aquela instrução SELECT, digitamos apenas:

```
select * from vw_venda;
```

id_venda	data_hora	veiculo	qtd	componente	preco_unitario	preco_total	fornecedor	cod_for
1	2022-04-01 00:00:00	Uno 2020	1	Velas de ignicao	600.00	600.00	Bosch	VL.BSH.UNO.2020
2	2022-04-01 00:00:00	Gol 2021	1	Jogo de cabos de vela	240.00	240.00	NGK	JC.NGK.GOL.2021
2	2022-04-01 00:00:00	Gol 2021	1	Palheta lado motorista	103.00	103.00	Dyna	LE.DYN.GOL.2021

Pontos de atenção

- Views não refletem automaticamente alterações realizadas nas tabelas. Se colunas forem adicionadas e necessitemos delas nas views, é preciso alterar a view manualmente.
- Não é possível criar índices para views. As views não possuem dados concretamente. Elas dependem das tabelas subjacentes. Logo, para otimizar views temos que adicionar índices nas tabelas empregadas.
- Você pode usar qualquer view em instruções SELECT, exatamente como faz com tabelas. Então, é totalmente possível termos VIEWS “dentro” de VIEWS.

DESCRIBE

Esse útil comando SQL exibe as colunas de uma tabela ou view, facilitando a construção ou manutenção de consultas e novas views.

Exemplo:

DESCRIBE vw_venda;

Field	Type	Null	Key	Default	Extra
id_venda	int	NO		0	
data_hora	datetime	NO		NULL	
veiculo	varchar(87)	YES		NULL	
qtd	int	NO		NULL	
componente	varchar(80)	NO		NULL	
preco_unitario	decimal(7,2)	NO		NULL	
preco_total	decimal(17,2)	NO		0.00	
fornecedor	varchar(50)	NO		NULL	
cod_forn	varchar(30)	NO		NULL	

EXPLAIN

Outra ferramenta bastante útil é a instrução EXPLAIN.

Normalmente utilizada por DBAs, ela “explica” como o SGBD processa uma instrução SELECT, INSERT, UPDATE ou DELETE. Dessa forma, é possível identificar onde índices são utilizados ou não, facilitando a definição de otimizações.

Exemplo:

```
explain select * from vw_venda;
```

id	select_type	table	partitions	type	possible_keys	key
1	SIMPLE	vd	NULL	index	PRIMARY	idx_data_venda
1	SIMPLE	i	NULL	ref	fk_item_venda_venda, fk_item_venda_estoque	fk_item_venda_venda
1	SIMPLE	f	NULL	index	PRIMARY	uk_nome
1	SIMPLE	e	NULL	eq_ref	PRIMARY, fk_estoque_componente, fk_estoque...	PRIMARY
1	SIMPLE	c	NULL	eq_ref	PRIMARY, uk_descricao	PRIMARY
1	SIMPLE	v	NULL	ref	PRIMARY, fk_veiculo_montadora	PRIMARY
1	SIMPLE	m	NULL	eq_ref	PRIMARY	PRIMARY

Note que nossa view sempre busca dados através de índices, ou seja, é uma query otimizada.

O que vimos nesse capítulo

Aprendemos a criar **VIEWS** para salvar nossas instruções **SELECT**.

Conhecemos a instrução **DESCRIBE** que exhibe a estrutura de tabelas e *views*.

Vimos que **EXPLAIN** pode ser empregado para identificar consultas lentas e os principais motivos, com a finalidade de otimizá-las.

Não é muito usual mencionar *describe* e *explain* ao falar a respeito de *views*, mas são duas ferramentas úteis, tanto para analisar *views* existentes quanto avaliar se são performáticas ou geraram necessidades de índices adicionais. Então, considere o momento certo, para ajudá-lo(a) a recordar-se dessas ferramentas quando são mais úteis.

"Daria tudo que sei pela metade do que ignoro."
René Descartes

Capítulo XII – Triggers

Quanto já vimos até aqui, não?

Bem, chegou o momento de resolver uma pendência relacionada com duas tabelas de nosso esquema. O valor total de cada item vendido e o total de uma venda.

Recordando: quando propus o esquema utilizado nesse guia, pedi para que definisse um campo “valor total” nas duas tabelas, indicando que inicialmente não seriam fornecidos nas inclusões, e, portanto, deveriam ser campos (colunas) opcionais (que aceitam null como valor).

Isso foi intencional; justamente para apresentar o valor dos *triggers* (gatilhos).

Triggers ou gatilhos, em português, são trechos de instruções SQL executados quando ocorrem eventos numa tabela de nosso banco de dados.

Sintaxe:

```
[DELIMITER caracteres]
CREATE TRIGGER nome_trigger
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON tabela
[FOR EACH ROW]
[BEGIN]
    instruções
[END caracteres]
[DELIMITER ; caracteres]
```

Quero esclarecer que a sintaxe pode conter mais elementos, e suportar funcionalidades diferentes, dependendo do SGBD. Mas essa sintaxe vale para 99% dos casos.

[DELIMITER caracteres]

A maioria dos SGBD define essa necessidade. Antes de codificar um ou vários triggers num script, devemos alterar o terminador padrão de instruções, de ponto-e-vírgula (;) para outra sequência qualquer. Por exemplo: ^^

CREATE TRIGGER nome_trigger;

Aqui inicia-se a definição do trigger em si, com a definição de um nome único no esquema.

{BEFORE | AFTER} {INSERT | UPDATE | DELETE}

Agora definimos quando o trigger é disparado: Antes ou após uma inclusão, alteração ou exclusão de registros.

ON tabela

Aqui indicamos a qual tabela o *trigger* relaciona-se.

FOR EACH ROW

Essa frase indica se o trigger será disparado apenas uma vez (quando não especificado) ou a cada registro afetado pela instrução (frase presente). Ou seja, um trigger sem a frase "FOR EACH ROW" disparará apenas uma vez, antes ou após a execução do INSERT, UPDATE ou DELETE. Sinceramente, nunca encontrei uso dessa forma. Em 100% das vezes encontrei triggers "FOR EACH ROW"; tanto que, **MySQL atualmente só suporta triggers FOR EACH ROW**, e a presença da frase é obrigatória nesse SGBD.

BEGIN

Indica o início do trecho de instruções que serão executadas quando o *trigger* for disparado.

instruções

Todas as instruções SQL que desejamos que sejam executadas quando o *trigger* for disparado.

Ao longo dessas instruções é normal ser necessário acessar os dados que estão sendo inseridos, alterados ou excluídos. Para diferenciar os dados existentes dos dados novos, dois prefixos são utilizados: **OLD** e **NEW**.

Supondo que uma tabela possua duas colunas numéricas denominadas "delta" e "valor". "delta" sempre armazena o quanto "valor" mudou na última atualização (update).

A instrução num trigger para manter "delta" sempre atualizada seria: **set NEW.delta = OLD.valor - NEW.valor;**

Várias instruções separadas por ponto-e-vírgula podem ser especificadas. Incluindo instruções de fluxo de processamento, como IF THEN ELSE.

[END caracteres]

Sinaliza o fim da definição do *trigger*.

[DELIMITER ;]

Restaura ponto-e-vírgula como o finalizador normal de instruções SQL.

Isso explicado, vamos tratar nossas pendências.

Pendência 1 – campo valor_total da tabela “venda”

“Primeiro, vamos criar o cadastro de **vendas**. Devem ser registradas as seguintes informações: Data da venda; vendedor que realizou a venda; nome do cliente, cujo preenchimento será opcional, e 50 posições são suficientes; CPE, também opcional, com até 11 posições e o valor total, com duas decimais e máximo de R\$ 500.000,00.

O valor total não será informado. Numa fase posterior o sistema calculará e preencherá esse campo. Logo, ele não pode ser obrigatório.

```
CREATE TABLE IF NOT EXISTS venda (  
  id INT NOT NULL AUTO_INCREMENT,  
  data_hora DATETIME NOT NULL,  
  vendedor_id INT NOT NULL,  
  nome_cliente VARCHAR(50),  
  cpf VARCHAR(11) DEFAULT 'NI',  
  valor_total DECIMAL(10,2),  
  PRIMARY KEY (id),  
  CONSTRAINT fk_venda_vendedor  
    FOREIGN KEY (vendedor_id) REFERENCES vendedor (id)  
    ON DELETE CASCADE ON UPDATE RESTRICT  
);
```

“Cada venda pode ser de um ou vários itens. Desses **itens vendidos** iremos registrar: o item de estoque vendido; a quantidade vendida e o preço unitário, com duas decimais.

Deve existir um campo de preço total, também com duas decimais.

De cada item o vendedor informará apenas o item de estoque e a quantidade. O preço unitário deverá ser obtido do estoque, e ***o preço total ficará em branco por agora.***”

```
CREATE TABLE IF NOT EXISTS item_venda (  
  id INT NOT NULL AUTO_INCREMENT,  
  venda_id INT NOT NULL,  
  estoque_id INT NOT NULL,  
  quantidade INT NOT NULL,  
  preco_unitario DECIMAL(7,2) NOT NULL,
```



```

preco_total    DECIMAL(9,2),
PRIMARY KEY (id),
CONSTRAINT fk_item_venda_venda
  FOREIGN KEY (venda_id) REFERENCES venda (id)
  ON DELETE RESTRICT ON UPDATE RESTRICT,
CONSTRAINT fk_item_venda_estoque
  FOREIGN KEY (estoque_id) REFERENCES estoque (id)
  ON DELETE RESTRICT ON UPDATE RESTRICT
);

```

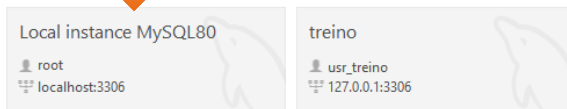
Precisamos de triggers que calculem o preço total de cada item de venda inserido ou alterado, atualizando o valor total da venda correspondente, inclusive quando itens de venda são excluídos.

Logo, são três *triggers*.

Mas, antes de prosseguir, deixe poupá-lo(a) de um tempo valioso. Há um bug no Workbench, e precisamos realizar a criação dos triggers como usuário **root**.

Volte para a tela principal do Workbench, e acesse como usuário root do banco local:

MySQL Connections ⊕ ⊞



Execute novamente o script de criação do esquema, para que tenhamos tabelas sem registros novamente (script **Cria_Esquema_Treino.sql**).

Em seguida, execute as instruções a seguir (se preferir, baixe a partir do link) para criar os triggers:



https://1drv.ms/u/s!AmhDenEg1Sjyhp5_S1np3pG6rt7o4Q?e=Uky5dN

```

USE treino;

DROP TRIGGER IF EXISTS item_venda_bi;
DROP TRIGGER IF EXISTS item_venda_bu;
DROP TRIGGER IF EXISTS item_venda_bd;

DELIMITER ^^

CREATE TRIGGER item_venda_bi
BEFORE INSERT
ON item_venda
FOR EACH ROW
BEGIN
    -- Calcula o preco total do item
    SET NEW.preco_total = NEW.quantidade *
    NEW.preco_unitario;

    -- Atualiza a venda relacionada
    UPDATE venda
    SET valor_total = valor_total + NEW.preco_total
    WHERE id = NEW.venda_id;
END ^^

CREATE TRIGGER item_venda_bu
BEFORE UPDATE
ON item_venda
FOR EACH ROW
BEGIN
    -- Desconta o valor total antigo do item da venda
    UPDATE venda SET valor_total = valor_total -
    OLD.preco_total WHERE id = OLD.venda_id;
    -- Calcula o novo valor total do item
    SET NEW.preco_total = NEW.quantidade *
    NEW.preco_unitario;
    -- Adiciona o valor total novo do item na venda
    UPDATE venda SET valor_total = valor_total +
    NEW.preco_total WHERE id = NEW.venda_id;

```

```
END ^^
```

```
CREATE TRIGGER item_venda_bd
BEFORE DELETE
ON item_venda
FOR EACH ROW
BEGIN
    -- Desconta o valor do item excluído da venda
    UPDATE venda
    SET valor_total = valor_total - OLD.preco_total
    WHERE id = OLD.venda_id;
END ^^
```

```
DELIMITER ;
```

Finalmente, execute novamente o script que alimenta nossas tabelas com dados (script **Popula_Esquema_Treino.sql**)

Se tudo correr como esperado, você observará os seguintes resultados:

```
SELECT * FROM item_venda LIMIT 10;
```

id	venda_id	estoque_id	quantidade	preco_unitario	preco_total
1	1	1	1	600.00	600.00
2	2	44	1	440.00	440.00
3	2	92	1	240.00	240.00
4	2	164	1	103.00	103.00
5	2	188	1	104.00	104.00
6	3	96	1	400.00	400.00
7	4	112	1	76.00	76.00
8	4	136	1	76.00	76.00
9	5	116	1	102.00	102.00
10	5	140	1	102.00	102.00

```
SELECT * FROM venda LIMIT 10;
```

id	data_hora	vendedor_id	nome_cliente	cpf	valor_total
1	2022-04-01 00:00:00	1	Celso	NULL	600.00
2	2022-04-01 00:00:00	2		NULL	887.00
3	2022-04-01 00:00:00	3		NULL	400.00
4	2022-04-01 00:00:00	1		NULL	152.00
5	2022-04-01 00:00:00	1		NULL	204.00
6	2022-04-02 00:00:00	2	Sr. Gilson	NULL	750.00
7	2022-04-02 00:00:00	4		NULL	1020.00
8	2022-04-02 00:00:00	3		NULL	220.00
9	2022-04-02 00:00:00	3	Aline	1100023009	158.00
10	2022-04-02 00:00:00	2		NULL	70.00

Observe que as colunas “preco_total” e “valor_total” das tabelas agora estão devidamente preenchidas. Os triggers funcionaram.

O pedido 2 é composto de 4 itens e totaliza 887,00.

Excluindo o item no valor de 240,00:

```
START TRANSACTION;
```

```
DELETE FROM item_venda WHERE venda_id = 2 AND id = 3;
```

```
SELECT * FROM venda LIMIT 10;
```

id	data_hora	vendedor_id	nome_cliente	cpf	valor_total
1	2022-04-01 00:00:00	1	Celso	NULL	600.00
2	2022-04-01 00:00:00	2		NULL	647.00
3	2022-04-01 00:00:00	3		NULL	400.00
4	2022-04-01 00:00:00	1		NULL	152.00
5	2022-04-01 00:00:00	1		NULL	204.00
6	2022-04-02 00:00:00	2	Sr. Gilson	NULL	750.00
7	2022-04-02 00:00:00	4		NULL	1020.00
8	2022-04-02 00:00:00	3		NULL	220.00
9	2022-04-02 00:00:00	3	Aline	1100023009	158.00
10	2022-04-02 00:00:00	2		NULL	70.00

Comprovamos que o valor total do pedido foi atualizado.

E se excluímos todos os itens desse pedido:

```
DELETE FROM item_venda WHERE venda_id = 2;  
SELECT * FROM venda LIMIT 10;
```

id	data_hora	vendedor_id	nome_cliente	cpf	valor_total
1	2022-04-01 00:00:00	1	Celso	NULL	600.00
2	2022-04-01 00:00:00	2		NULL	0.00
3	2022-04-01 00:00:00	3		NULL	400.00
4	2022-04-01 00:00:00	1		NULL	152.00
5	2022-04-01 00:00:00	1		NULL	204.00
6	2022-04-02 00:00:00	2	Sr. Gilson	NULL	750.00
7	2022-04-02 00:00:00	4		NULL	1020.00
8	2022-04-02 00:00:00	3		NULL	220.00
9	2022-04-02 00:00:00	3	Aline	1100023009	158.00
10	2022-04-02 00:00:00	2		NULL	70.00

Comprovamos que o total da venda vai a zero.

ROLLBACK;

Só queríamos testar...

O que vimos nesse capítulo

Aprendemos a função dos TRIGGERS de tabelas, aplicando na prática os conceitos, e assim, tratando as últimas pendências quanto ao esquema proposto.

Vimos os triggers em ação, atualizando automaticamente colunas de nossas tabelas por nós.

Eu tentei 99 vezes e falhei, mas na centésima tentativa eu consegui; nunca desista de seus objetivos mesmo que esses pareçam impossíveis, a próxima tentativa pode ser a vitoriosa.

Albert Einstein

Capítulo XIII – Stored Procedures

No capítulo anterior vimos o SGBD executar códigos automaticamente, quando eventos ocorrem em nossas tabelas.

Mas, e se for necessário executar várias instruções sem necessariamente ter ocorrido algo com nossos dados?

Para isso existem as *Stored Procedures* (procedimentos armazenados). Trechos de instruções que somente executam se forem explicitamente invocadas.

Um bom exemplo seria um processo de atualização mensal do total de vendas do mês anterior, sempre executado no primeiro dia útil do mês atual. Algo que só deve executar uma vez por mês.

Para atender esse cenário, podemos criar uma *STORED PROCEDURE*, e executá-la apenas no primeiro dia útil.

CREATE PROCEDURE

A sintaxe de criação de uma *STORED PROCEDURE* é:

```
[DELIMITER caracteres]
CREATE PROCEDURE nome_procedure (params...)
BEGIN
    Instruções;
END caracteres
[DELIMITER ;]
```

params é uma lista opcional de parâmetros que a procedure recebe e/ou devolve.

Devolve???

Sim, devolve. Diferente de funções em Java, C ou Python, procedures SQL podem usar os parâmetros tanto para receber dados, como para devolver dados.

Para exemplificar, vamos criar uma *PROCEDURE* que recebe um número de venda qualquer, e nos devolve o valor total.

```
USE treino;  
DROP PROCEDURE IF EXISTS valor_venda;  
DELIMITER $$  
CREATE PROCEDURE valor_venda(IN vvenda INT, OUT vttotal  
DECIMAL(10,2))  
BEGIN  
    SELECT valor_total INTO vttotal  
    FROM venda  
    WHERE id = vvenda;  
END $$  
DELIMITER ;
```

CALL

Executamos a procedure através da instrução CALL:

```
CALL valor_venda(3, @var);  
SELECT @var;
```

DROP PROCEDURE

Para excluir uma procedure existente, utilizamos DROP PROCEDURE.

```
DROP PROCEDURE valor_venda;
```


O que vimos nesse capítulo

Aprendemos como criar procedimentos armazenados (STORED PROCEDURES), que podemos executar à medida de nossas necessidades.

Vimos como executar os procedimentos armazenados.

Aprendemos a excluir os procedimentos armazenados.

“Você não é derrotado quando perde. É derrotado quando desiste”

Dr. House (personagem)

Capítulo XIV – Tópicos finais

Antes de ver os itens desse capítulo, quero falar um pouco sobre colocar ou não colocar regras de negócio num banco de dados, na forma de *triggers* ou *stored procedures*.

Se você refletir um pouco a respeito, chegará à conclusão de que a linguagem SQL nos permite, através de seus recursos, codificar boa parte da lógica de negócios diretamente no banco de dados.

Asseguro a você, meu/minha amigo(a), que discussões sobre se isso deve ou não ser feito é antiga e vasta, repleta de argumentos favoráveis e contrários.

Um forte argumento para fazer, é a mitigação de erros na manutenção dos sistemas. Um forte argumento contra, é a dependência de fornecedor de SGBD que isso provoca, pois como vimos, sempre há um detalhe de implementação que é particular a cada um.

De fato, penso que não é uma questão de preto e branco. Há infinitos cinzas nesse cenário. Erros de manutenção podem ocorrer não somente na manutenção dos sistemas, mas também na manutenção dos scripts SQL. Troca de SGBD é historicamente raro, não ocorre a todo instante.

Para mim, a decisão deve levar em conta vários fatores:

- Nível de experiência das equipes;
- Burocracias de atualização de BD e de sistemas da empresa;
- Quantidade de sistemas que devem implementar uma data regra *versus* riscos em se colocar no BD;

- Consumo de CPU dos servidores de bancos de dados pelo processamento das regras;
- Possibilidade ou não de informar o usuário se o SGBD rejeitar uma operação por restrições decorrentes das regras;
- Outras mais.

Logo, cada empresa precisa refletir e definir “suas regras do jogo”.

Isso posto, vamos ao conteúdo principal do capítulo...

Coalesce()

Deixei intencionalmente essa função para o final, dada sua relevância.

A função **coalesce()** é utilizada quando precisamos substituir qualquer valor **null** por outro valor.

Mas por que isso é tão relevante?

Porque **null** pode provocar efeitos colaterais de deixar qualquer *designer* de banco de dados maluco. Principalmente em expressões matemáticas e instruções IF.

Por exemplo, nossa tabela de vendas possui um campo de total. Como definido, é um campo opcional, cujo valor será calculado por nossas *triggers*. Conhecedor desses efeitos colaterais envolvendo **null**, na definição do campo em minha solução proposta adicionei **DEFAULT 0**. Ou seja, o SGBD sempre gravará o valor 0.00 para novos registros.

Lembra-se da instrução de atualização desse campo no trigger que é disparado quando um item de venda é inserido?

UPDATE venda

```
SET valor_total = valor_total + NEW.preco_total  
WHERE id = NEW.venda_id;
```

Solicitamos que o SGBD grave no campo **valor_total** da tabela venda o resultado de sua soma com o valor do campo **preco_total** do item de venda inserido.

Como temos a cláusula DEFAULT na tabela vendas, para a primeira inclusão de um item de vendas de valor 100,00, o cálculo seria:

$\text{SET valor_total} = 0.00 + 100.00$

Ou seja, $\text{valor_total} = 100.00$

Agora, se não houvesse a cláusula DEFAULT, **valor_total** armazenaria null inicialmente. Logo:

$\text{SET valor_total} = \text{null} + 100.00$

E teríamos $\text{valor_total} = \text{null}$

Qualquer valor somado, dividido, multiplicado ou subtraído de null resulta null!

Como designer de BD, você ficaria procurando o motivo da sua trigger não funcionar. **É frustrante.**

E para ser honesto, o código ideal para o update de nossa trigger seria esse:

```
UPDATE venda
SET valor_total = coalesce(valor_total,0) +
coalesce(NEW.preco_total, 0)
WHERE id = NEW.venda_id;
```

Pois assim garantiríamos que nunca valor_total seria atualizado com NULL numa inclusão de item de venda.

Para comprovar esses efeitos de NULL, pode testar esses SELECT no Workbench:

```
SELECT null + 100;
```

Resulta null

```
SELECT CASE WHEN null = null THEN 1 ELSE 0 END;
```

Resulta 0

Sequences e Generators

Você aprendeu a usar colunas autoincrementadas, correto?

Bem, os SGBD relacionais oferecem um tipo de objeto denominado SEQUENCE (ou GENERATORS como alguns os chamam), que são geradores automáticos de sequências numéricas. Os campos autoincrementados utilizam esses objetos nos bastidores.

MySQL não possui essa instrução, mas em Postgres a sintaxe é:

```
CREATE SEQUENCE nome_seq
INCREMENT BY passo
START WITH valor_inicial;
```

Para obter o próximo valor e incrementar ao mesmo tempo:

```
select nextval('nome_seq');
```

E para obter o valor atual, sem incrementar:

```
select currval('nome_seq');
```

Colunas BLOB e CLOB

Não abordei nesse guia a possibilidade de armazenar imagens e outros objetos binários em tabelas do banco de dados.

Isso é possível, e as estruturas que suportam esses tipos de dados são as colunas BLOB e CLOB.

BLOB é indicada para objetos binários, quanto CLOB é indicada para textos longos, como Scripts e outros.

Caso queira ver como isso é feito, por favor, recorra ao guia do fornecedor de SGBD que necessitar ou preferir.

Dada a carga de dados envolvida nessas operações, sempre há muitas informações e melhores práticas descritas por cada fornecedor.

Particionamento

Nosso esquema de exemplo é pequeno, com poucos dados. Mas num banco de dados de produção, isso quase nunca é verdade.

Questões de performance surgem a todo instante, e podem estar ligadas ao hardware, performance da rede ou arquitetura do banco de dados.

Um recurso disponibilizado por praticamente todo fornecedor de bancos de dados relacionais é a capacidade de particionamento automático de tabelas.

Pode-se, por exemplo, definir que uma tabela seja particionada pelo mês e ano de uma coluna tipo DATA qualquer; ou sempre que atingir 1.000.000 de registros, e outras configurações.

No MySQL essa definição é realizada através da cláusula **PARTITION BY** da instrução **CREATE TABLE**.

A vantagem do particionamento, é que os dados da tabela são armazenados em arquivos distintos em disco, que podem ser direcionados para diferentes unidades de armazenamento. Assim, o SGBD pode tirar proveito de leituras e escritas em paralelo, o que colabora para melhor performance.

Alguns SGBD incluem ainda, o conceito de **TABLESPACES**, ou áreas de armazenamento para várias tabelas. Nesse caso, um banco de dados pode possuir várias *tablespaces* (arquivos em disco), e tabelas podem ter seus dados distribuídos em várias dessas *tablespaces*; tudo definido pelo DBA ao criar o banco e/ou os esquemas.

Backup e Restore

Dados que armazenamos num SGBD devem ser importantes, não acha?

Dados importantes requerem backup. Afinal, perder informação, seja qual for o motivo, resulta em prejuízos.

Todo SGBD disponibiliza mecanismos para realização de backups, sem ou com paradas parciais ou totais do banco de dados.

Não vamos aqui exercitar ou ir mais afundo nisso, mas deixe-me explicar brevemente três conceitos populares relacionados às modalidades de backup:

COLD

É o backup com o banco de dados ou um esquema bloqueado para consultas e alterações. Um esquema ou todo o banco de dados é colocado num modo específico pelo DBA, onde conexões não são permitidas, e assim, uma cópia exata dos dados no momento da paralisação pode ser feita para outros servidores ou dispositivos de armazenamento. Em geral, essas cópias são gravadas em fitas ou outras mídias mais baratas que os discos rígidos, para arquivamento de longo prazo.

STANDBY

O banco de dados é configurado para gerar instantâneos contendo todas as instruções DDL e instruções DML de inserção e alteração de dados. De tempos em tempos, pequenos arquivos com nomes automáticos são gerados. Um segundo servidor de banco de dados é configurado para receber esses logs. Então, uma tarefa coleta os logs do servidor principal e envia para o servidor secundário, que lê os arquivos e aplica as mesmas instruções que ocorreram no servidor principal, na mesma ordem em que ocorreram. Isso resulta num servidor pronto para assumir as funções do servidor principal, em caso de pane, com mínima perda de dados.

HOT STANDBY

Trata-se da técnica COLD STANDBY, mas com o servidor secundário configurado para responder a instruções SELECT, distribuindo-se assim, a carga de atendimento de pesquisas.

Clusters

Além da necessidade de manter cópias de segurança de dados, manter a performance à medida que o uso de um banco de dados cresce é essencial. A demanda crescente nesse sentido fez surgir os clusters de bancos de dados. Vários servidores atendendo as demandas de consulta e escrita de dados simultaneamente, dividindo a carga de trabalho ao mesmo tempo em que garantem consistência.

Oracle, Microsoft SQL Server, Postgres e MySQL oferecem soluções de cluster. Praticamente todo provedor oferece.

Obviamente esse tópico vai além do escopo desse pequeno guia, mas não poderia de deixar de citar aqui.

O que vimos nesse capítulo

Uma breve reflexão sobre colocar ou não regras de negócio em bancos de dados.

Conhecemos a importante função **coalesce()** e algumas armadilhas envolvendo **NULL**.

Aprendemos sobre *sequences/generators* e como utilizá-los.

Os tipos de dados **BLOB** e **CLOB** para armazenar dados grandes foram brevemente apresentados.

Tivemos uma visão geral sobre **particionamento**.

Uma visão conceitual sobre *backup* e *restore* e os *clusters* de bancos de dados foi apresentada.

“Quando você deseja o bem, o bem te deseja também”

Frases do bem

Capítulo XV – Conexão SGBD x Programas

Se você seguiu o guia como apresentado, praticamente usou o tempo todo o utilitário Workbench para lidar com nosso banco de dados. Isso foi proposital. Lidar diretamente com o banco de dados, sem usar uma IDE como Visual Studio Code, Eclipse, Jupiter Notebook etc., tira do caminho detalhes relacionados com essas ferramentas. Assim, você pôde focar puramente nas questões do banco de dados e detalhes da linguagem SQL.

Mas claro, profissionalmente, dependendo de sua função, haverá necessidade de utilizar o banco de dados em programas ou outras ferramentas, como o *Microsoft Power BI*, por exemplo.

Nestes casos, é preciso fazer “seja lá o que for” conversar com o SGBD.

Drivers

Os programas, ou bibliotecas que permitem uma solução qualquer conectarem-se com o banco de dados são denominadas *drivers*. Os tipos de driver mais populares são o ODBC (*Open Database Connectivity*) e o JDBC (*Java Database Connectivity*).

Esses drivers são instalados nos computadores que precisam comunicar-se diretamente com o banco de dados.

A conexão é realizada através de uma “*connection string*”, onde são informados:

- O endereço ou nome do servidor onde o SGBD executa;
- A porta TCP/IP de conexão;

- O usuário de conexão;
- A senha do usuário de conexão.

Se você voltar a tela inicial do Workbench e clicar no ícone de ferramenta (manutenção das conexões), verá esses parâmetros facilmente:

Parameters	SSL	Advanced
Hostname:	<input type="text" value="127.0.0.1"/>	Port: <input type="text" value="3306"/> Name or IP address of the server host - and TCP/IP port.
Username:	<input type="text" value="root"/> Name of the user to connect with.	
Password:	<input type="button" value="Store in Vault ..."/> <input type="button" value="Clear"/> The user's password. Will be requested later if it's not set.	

A *string* de conexão JDBC para nosso banco de dados treino é:

`jdbc://127.0.0.1:3306/treino`

A instrução Java para abrir uma conexão com nosso banco seria:

```
Connection conn = DriverManager.getConnection(
“jdbc://127.0.0.1:3306/treino”, “usr_treino”, “<senha>”);
```

Onde <senha> teríamos que substituir pela aquela que definimos.

Em Python, muito similar (utilizando Pandas):

```
import mysql.connector as sql
import pandas as pd

db_connection = sql.connect(host='127.0.0.1',
database='treino', user='usr_treino', password='<senha>')
```

O que vimos nesse capítulo

Aprendemos sobre como os programas utilizam *drivers* para estabelecerem conexão com um SGBD.

Exemplos de *strings* de conexão Java e Python foram fornecidos.

Aprendemos que os dados básicos de conexão são:

- ✓ O endereço ou nome do servidor;
- ✓ A porta de comunicação TCP/IP a utilizar;
- ✓ O nome de usuário a empregar;
- ✓ A senha do usuário de conexão.

“Não deixe o ruído das opiniões dos outros abafar a sua própria voz interior”

Steve Jobs

Capítulo XVI – Conclusão

Sistemas Gerenciadores de Bancos de Dados Relacionais são ferramentas poderosas e repletas de recursos. Sua enorme gama de versões, de gratuitas as que suportam multinacionais em regimes 24x7 constituem um ecossistema gigantesco.

Tentei construir um caminho fácil para iniciar e evoluir gradativamente na compreensão dessa tecnologia, iniciando pelo básico, adicionando detalhes à medida em que se tornaram necessários, e penso, proporcionando momentos de fácil exercício daquilo que foi aprendido em cada capítulo.

Penso que esse caminho é coerente, e de ajuda para muitos, que de algum modo, sentiram dificuldades em outras abordagens.

Se fui bem sucedido, ficaria muito agradecido se pudesse deixar seu like em meu blog, no post em que compartilhei esse pequeno trabalho.

E espero, de que algum modo, o tempo que dedicou aqui resulte em mudanças positivas em sua jornada.

Deixo aqui meus agradecimentos, e um forte abraço, na esperança de que possamos iniciar e manter contato.

Alexandre Rozante.