Apostila do curso

MySQL: Do Zero à consulta de base de dados pública

Celso Y. Ishida

2025

Apresentação

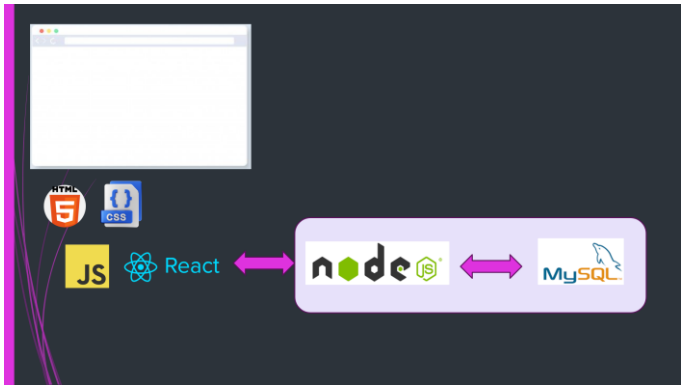
Olá! Bem-vindo ao mundo do Banco de Dados. Este livro foi criado para quem deseja manipular grande bases de dados mas não é da área de TI.

Vídeo: boas vindas

Se você não é da área de tecnologia, e deseja analisar base de dados, muito facilitado com

Este livro, revisado com auxílio de IA, resume os conceitos principais da linguagem e é complementado com os exercícios do curso de MySQL ministrado pelo professor Celso Yoshikazu Ishida e os vídeos indicados pelos QrCode. Os exercícios aparecem na ordem crescente de aprendizagem. É importante que você leia o livro e faça os exercícios na ordem, e volte para continuar o aprendizado.

Escolhemos o banco de dados MySQL, um dos SGBD (Sistema de Gerenciamento de Banco de Dados) relacional mais utilizados no mundo para a programação web. Mas já vamos entender como funciona a combinação de tecnologias que estamos utilizando em nossos projetos.



qrmysql001

ÍNDICE

Sumário

Introdução

Aprender um banco de dados relacional é aprender a linguagem SQL e sua sintaxe. Neste curso iremos partir de uma organização não recomendada no Excel até termos informações tentando garantir consistência, eliminar a duplicação (evita redundância) e integridade dos dados. Com o Excel é possível até garantir uma boa organização, atingindo a 3FN (normalização para garantir qualidade).

qrmysql002 Organização dados com MS Excel

A planilha eletrônica não é ideal para todos os requisitos que se espera na organização de um banco de dados, por isso, surge o SGBD que é o Sistema de Gerenciamento de Banco de Dados. O SGBD vai garantir, principalmente, segurança e facilidade para consulta e manipulação dos dados, ajudando a você transformar dados em informação e conhecimento. Já na primeira aula você poderá aprender os passos básicos para uso do SGBD para análise de dados, e descobrir que não precisa ser da área de TI para poder manipular as informações com o MySQL.

E vamos começar a nossa jornada. Veja os tópicos que serão abordados neste livro.

Tópicos importantes

- Consulta Select na prática: veremos na prática como selecionar as informações que você deseja?
- Conceitos de Banco de dados: SQL, BD, SGBD, Tabela, Coluna, Linha, Chaves.
- Data Definition Language (DDL):
    - criação, alteração e remover
    - Tipos de Dados: Entendimento dos principais tipos de dados do MySQL (INT, VARCHAR, TEXT, DATE, DECIMAL, BOOLEAN, etc.) e quando usá-los.
- Data Manipulation Language (DML): CRUD
    - INSERT INTO: Inserção de novos registros em tabelas.
    - UPDATE: Modificação de registros existentes.
    - DELETE FROM: Exclusão de registros.
- Conceitos do Select
    - SELECT: Recuperação de dados de tabelas.
    - Cláusulas WHERE: Filtragem de registros com base em condições.
    - Operadores de Comparação e Lógicos: =, >, <, >=, <=, <>, AND, OR, NOT.
    - ORDER BY: Ordenação de resultados.
    - LIMIT: Limitação do número de resultados.
    - Funções de Agregação: COUNT, SUM, AVG, MIN, MAX.
    - GROUP BY e HAVING: Agrupamento de resultados e filtragem de grupos.
- Consultas Avançadas e Junções (JOINs)
    - Junções (JOINs):
        - INNER JOIN: Recuperação de registros que têm correspondência em ambas as tabelas.
        - LEFT JOIN (ou LEFT OUTER JOIN): Recuperação de todos os registros da tabela da esquerda e os registros correspondentes da tabela da direita.
        - RIGHT JOIN (ou RIGHT OUTER JOIN): Recuperação de todos os registros da tabela da direita e os registros correspondentes da tabela da esquerda.

    - Subconsultas (Subqueries): Utilização de consultas dentro de outras consultas.
    - UNION e UNION ALL: Combinando resultados de múltiplas consultas.
- Restrições e Índices
    - Integridade Referencial: Como as chaves estrangeiras garantem a consistência dos dados.
    - Restrições: PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, DEFAULT, CHECK.
    - Índices: O que são, como funcionam e quando criá-los para melhorar o desempenho das consultas.
- Funções Comuns do MySQL
    - Funções de String: CONCAT, LENGTH, SUBSTRING, UPPER, LOWER.
    - Funções Numéricas: ROUND, CEIL, FLOOR, ABS.
    - Funções de Data e Hora: NOW, CURDATE, CURTIME, DATE_FORMAT, DATEDIFF.
- Modelagem de banco de dados:
    - Introdução ao conceito de modelagem,
    - Entidades, Atributos e relacionamentos
    - diagramas ER (Entidade-Relacionamento) simples e como traduzir um modelo para um banco de dados.
- Stored Procedures
    - O que são Stored Procedures: Explicação do conceito de stored procedures como conjuntos de instruções SQL armazenadas no banco de dados.
    - Sintaxe Básica:
    - Parâmetros:
    - Variáveis: Declaração e uso de variáveis dentro de stored procedures.
    - Controle de Fluxo:
    - Exemplos Práticos: Criação e execução de stored procedures simples para realizar tarefas comuns.
    - Gerenciamento de Stored Procedures:
- Administração
    - Importação de base de dados pública
    - backup
    - Segurança:
        - Concessão e revogação de privilégios para executar stored procedures.

Com estas aulas espera-se que você consiga programar com banco de dados e/ou manipular as informações para análise de dados pública.

Consulta Select na prática

Vamos começar pelo ponto crucial para qualquer tomador de decisões: a **seleção e recuperação de informações** (ou *Information Retrieval*). É aqui que transformamos dados brutos em **conhecimento útil**, que é a base para as escolhas estratégicas, justamente para responder nossas questões com relação a uma base de dados específica.

Já vimos que podemos organizar nossas informações com o MS Excel, agora vamos aproveitar a organização feita para fazer uma analogia com um banco de dados relacional:

- Seu **arquivo** no Excel é como um **banco de dados (database)** no MySQL.
- Cada **planilha** dentro do seu arquivo Excel é como uma **tabela** no MySQL.
- As **colunas** da sua planilha são, adivinhe só, as **colunas** da sua tabela no MySQL.

É tudo sobre organizar os dados de forma lógica para que possamos encontrá-los e analisá-los facilmente.

Para colocar a mão na massa, vamos usar um conjunto de dados real: as movimentações (contratações e demissões) do **Novo CAGED** de um mês específico. Com ele, vamos responder a algumas perguntas interessantes usando um comando superimportante do MySQL: o **SELECT**.

O comando `SELECT` é a sua ferramenta para "pedir" ao banco de dados as informações que você quer ver. Este comando deve ser escrito na forma:

```
SELECT [coluna(s)]
FROM [Tabela(s)]
WHERE [restrição]
```

Vamos desmembrar isso:

- **SELECT [coluna(s)]**: Aqui você diz quais colunas você quer ver separadas por vírgulas. Se quiser ver todas as colunas, pode usar um asterisco (*).
- **FROM [Tabela(s)]**: Aqui você indica de qual tabela você quer buscar as informações.
- **WHERE [restrição(ões)]**: Esta parte é opcional, mas muito útil! Com ela, você pode filtrar os resultados, ou seja, dizer ao MySQL para mostrar apenas as linhas que atendem a uma determinada condição.
- **Order by [coluna(s)]**: Aqui você diz quais colunas serão importantes para a ordenação, separadas por vírgulas. Pode-se utilizar o número da ordem das colunas.

Obs: Where e Order by são opcionais.

Agora, vamos às nossas perguntas e como usar o `SELECT` para respondê-las:

Quais as cidades, seção, subclasse, idades e salários do Paraná que com foram contratados com salário maior do que 100.000, ordenados pela idade e depois cidade?

Para responder a essa pergunta, precisaríamos de uma tabela com informações sobre seção, subclasse, idades e salários. A restrição seria o saldo de movimentação como contratação e salário maior do que 100.000.

Quantos Cientistas de Dados foram admitidos em Curitiba?

Para responder a essa pergunta, precisaríamos de uma tabela com informações sobre as admissões, incluindo o cargo e a cidade. A restrição seria para o cargo "Cientista de Dados" e para a cidade "Curitiba".

Qual a média salarial dos admitidos que um profissional de gestão da informação pode atuar?

Aqui, teríamos que identificar os cargos que um profissional de gestão da informação pode atuar e então calcular a média salarial para esses cargos específicos entre os admitidos.

Qual a média salarial de programadores?

Neste caso, a restrição seria o cargo "Programador" e o resultado seria a média dos salários dos admitidos com esse cargo.

Um mestre ganha mais do que um graduado?

Para responder a essa pergunta, precisaríamos comparar a média salarial de pessoas com o nível de escolaridade "Mestrado" com a média salarial de pessoas com o nível de escolaridade "Graduação".

Qual a média salarial de quem terminou o ensino médio e não está fazendo curso superior?

Aqui, a restrição seria para pessoas com escolaridade "Ensino Médio Completo" e que não estejam cursando ensino superior.

Não se preocupe se as perguntas parecem um pouco complexas agora. O importante é entender que o comando SELECT é a chave para extrair as informações que você precisa de um banco de dados, e que as cláusulas FROM e WHERE te ajudam a especificar exatamente de onde e quais dados você quer ver. Nos próximos capítulos, vamos ver exemplos práticos de como construir esses comandos para obter as respostas desejadas.

Vamos visualizar como se faz as consultas para responder as perguntas acima?

Conseguiu ver a facilidade de execução e seleção das informações? Agora vamos ver em detalhes sobre o comando SELECT.

O Comando SELECT: A Chave para Buscar Informações

Pense no comando **SELECT** como a sua **ferramenta principal para "perguntar" ao banco de dados o que você quer ver**. É com ele que você escolhe quais informações (Colunas)

deseja visualizar, de qual "tabela" (lembra do exemplo da planilha do Carro?) elas devem vir, e até mesmo quais "linhas" (registros) específicas você quer considerar.

Em termos simples, o SELECT é o que te permite **selecionar dados** de uma ou mais tabelas. Ele é a base de quase toda interação com um banco de dados quando o seu objetivo é extrair informações para análise ou visualização.

O comando SELECT é, sem dúvida, o mais utilizado em SQL. Ele permite que você recupere dados das suas tabelas.

Observação: as tabelas utilizadas nos exemplos estão no final da apostila.

Sintaxe Básica:

```
SELECT Colunas
FROM  Tabelas;
```

Neste comando, no lugar de [Colunas] pode ser qualquer coluna da tabela separado por vírgulas ou apenas o asterisco (*) indicando todas as colunas.

A seguir exemplos e suas explicações[1]:

`SELECT * FROM Movimentacao;`	Retorna todos os registros da tabela Movimentacao mostrando os valores de todas suas colunas
`SELECT SALDOMOVIMENTACAO, MUNICIPIO, SALARIO FROM Movimentacao;`	Seleciona todos os registros retornando as colunas: salmovimentacao, cidade e salario.

Existe um complemento opcional do comando, mas muito utilizado, que restringe os registros de acordo com condições incluídas depois da palavra 'WHERE'

`SELECT racacor, graudeinstrucao FROM Movimentacao WHERE municipio = 410690 and cbo2002ocupacao = 211220;`	Seleciona os registros de Curitiba e da ocupação de 'Cientista de Dados', mostra as colunas Raça e cor e o grau de instrução.

Existe um complemento opcional do comando que ordena os registros de acordo com os valores das colunas número das colunas de retorno: order by [colunas] [asc/desc]

`SELECT SEXO, IDADE, SALARIO FROM Movimentacao ORDER BY idade, sexo;`	Seleciona todos os registros, mostra as colunas sexo, idade, salario e mostra os registros na ordem crescente de idade e depois sexo

A seguir outros exemplos de consulta e mais detalhes:

```
-- Selecionar todos os autores
SELECT * FROM autores;

-- Selecionar apenas o título e ano de publicação dos livros
SELECT titulo, ano_publicacao FROM livros;
```

[1] A definição das tabelas que são utilizadas como exemplos estão no final do livro

Cláusulas WHERE: Filtrando Registros

A cláusula WHERE permite que você filtre os registros retornados, selecionando apenas aqueles que satisfazem uma ou mais condições.

```
SELECT colunas FROM nome_da_tabela WHERE condicao;
```

A condição é formada pela coluna comparação e valor e, se tiver mais colunas um operador lógico. Exemplos de condições:

```
Nome = 'Letícia'
and nacionalidade = 'brasileira'
```

Operadores de Comparação:

- =: Igual a
- >: Maior que
- <: Menor que
- >=: Maior ou igual a
- <=: Menor ou igual a
- <> ou !=: Diferente de
- LIKE: Busca por padrões (usado com % para zero ou mais caracteres, _ para um único caractere).
- IN: Verifica se um valor está em uma lista de valores.
- BETWEEN: Verifica se um valor está dentro de um intervalo.
- IS NULL: Verifica se um valor é nulo.
- IS NOT NULL: Verifica se um valor não é nulo.

Operadores[1] Lógicos:

- AND: Retorna TRUE se todas as condições forem verdadeiras.
- OR: Retorna TRUE se pelo menos uma das condições for verdadeira.
- NOT: Inverte o resultado de uma condição (TRUE se for FALSE, e vice-versa).

Exemplos com WHERE:

```
-- Selecionar livros publicados depois de 1900
SELECT titulo, ano_publicacao FROM livros WHERE ano_publicacao > 1900;

-- Selecionar autores brasileiros
SELECT nome, nacionalidade FROM autores WHERE nacionalidade = 'Brasileira';

-- Selecionar livros cujo título começa com 'A'
SELECT titulo FROM livros WHERE titulo LIKE 'A%';

-- Selecionar livros publicados entre 1850 e 1950
SELECT titulo, ano_publicacao FROM livros WHERE ano_publicacao BETWEEN 1850 AND 1950;

-- Selecionar autores que não são brasileiros
SELECT nome, nacionalidade FROM autores WHERE NOT nacionalidade = 'Brasileira';
-- Ou
SELECT nome, nacionalidade FROM autores WHERE nacionalidade <> 'Brasileira';

-- Selecionar autores brasileiros nascidos antes de 1900
SELECT nome, data_nascimento FROM autores WHERE nacionalidade = 'Brasileira' AND
data_nascimento < '1900-01-01';

-- Selecionar livros de Machado de Assis OU de Clarice Lispector (usando os IDs 1 e 2, respectivamente)
SELECT titulo, id_autor FROM livros WHERE id_autor = 1 OR id_autor = 2;
-- Ou, de forma mais elegante
SELECT titulo, id_autor FROM livros WHERE id_autor IN (1, 2);
```

```
SELECT colunas FROM nome_da_tabela WHERE condicao ORDER BY coluna_ordenacao [ASC|DESC];

-- Ordenar livros por ano de publicação, do mais antigo para o mais novo
SELECT titulo, ano_publicacao FROM livros ORDER BY ano_publicacao ASC;

-- Ordenar autores por nome, em ordem alfabética inversa
SELECT nome FROM autores ORDER BY nome DESC;

-- Ordenar livros por autor (ID) e depois por título
SELECT id_autor, titulo FROM livros ORDER BY id_autor ASC, titulo ASC;
```

LIMIT: Limitando o Número de Resultados

A cláusula LIMIT é usada para restringir o número de registros retornados por uma consulta. É muito útil para paginação ou para obter apenas os "top N" resultados.

```
SELECT colunas FROM nome_da_tabela LIMIT numero_de_registros OFFSET inicio;
-- Ou
SELECT colunas FROM nome_da_tabela LIMIT inicio, numero_de_registros;
```

- numero_de_registros: Quantos registros retornar.
- inicio: Opcional. Onde começar a contagem (o primeiro registro é 0).

```
-- Retornar os 3 primeiros livros
SELECT titulo FROM livros LIMIT 3;

-- Retornar os 2 livros a partir do 3º (índice 2)
SELECT titulo FROM livros LIMIT 2 OFFSET 2;
-- Ou
SELECT titulo FROM livros LIMIT 2, 2; -- (primeiro é o OFFSET, segundo é o LIMIT)
```

Funções de Agregação

Funções de agregação realizam cálculos em um conjunto de linhas e retornam um único valor.

- COUNT(): Conta o número de linhas.
- SUM(): Calcula a soma dos valores de uma coluna numérica.
- AVG(): Calcula a média dos valores de uma coluna numérica.
- MIN(): Retorna o menor valor de uma coluna.
- MAX(): Retorna o maior valor de uma coluna.

```
-- Contar o total de livros
SELECT COUNT(*) FROM livros;

-- Encontrar o ano de publicação mais antigo
SELECT MIN(ano_publicacao) FROM livros;

-- Encontrar o ano de publicação mais recente
SELECT MAX(ano_publicacao) FROM livros;
```

GROUP BY e HAVING: Agrupamento e Filtragem de Grupos

GROUP BY agrupa linhas que têm os mesmos valores em uma ou mais colunas, permitindo que você aplique funções de agregação a cada grupo. HAVING é como WHERE, mas filtra grupos após a agregação.

```
SELECT colunas_agrupadas, FUNCAO_AGREGACAO(coluna)
```

```
FROM nome_da_tabela
GROUP BY colunas_agrupadas
HAVING condicao_do_grupo;

-- Contar quantos livros cada autor tem
SELECT id_autor, COUNT(id_livro) AS total_livros
FROM livros
GROUP BY id_autor;

-- Contar quantos livros cada autor tem, mas apenas para autores com mais de 1 livro
SELECT id_autor, COUNT(id_livro) AS total_livros
FROM livros
GROUP BY id_autor
HAVING total_livros > 1;

-- Média de anos de publicação por autor (se você tiver valores mais variados)
SELECT id_autor, AVG(ano_publicacao) AS media_ano_publicacao
FROM livros
GROUP BY id_autor;
```

Exercícios de seleção de informações

Agora, que tal procurar algumas informações sobre empregos? Mãos à obra

Resumindo e Agrupando Dados com MySQL

Neste capítulo vamos utilizar a tabela Movimentacao - Novo CAGED). Imagine uma grande planilha com milhões de registros de admissões e desligamentos em empresas no Brasil. Como responder perguntas como:

- Quantas pessoas foram admitidas por estado?

- Qual é a média salarial por grau de instrução?

- Em que faixa etária o salário médio é mais alto?

- Quantas movimentações envolvem aprendizes?

Tudo isso é possível com funções de agregação no MySQL, GROUP BY e HAVING. Vamos por partes.

1. Funções de Agregação

Essas funções ajudam a resumir informações de várias linhas em apenas uma:

Função	Para que serve
COUNT()	Conta o número de registros
SUM()	Soma valores numéricos
AVG()	Calcula a média
MIN()	Retorna o menor valor
MAX()	Retorna o maior valor

Exemplo 1: Total de admissões

```
SELECT COUNT(*) AS total_admissoes
FROM Movimentacao
WHERE saldomovimentacao = 1;
```

2. GROUP BY: Agrupar para Analisar

Com GROUP BY, conseguimos agrupar os dados por uma categoria — como sexo, estado ou grau de instrução — e aplicar as funções acima em cada grupo.

```
SELECT graudeinstrucao, COUNT(*) AS total_desligamentos
FROM Movimentacao
WHERE saldomovimentacao = -1
GROUP BY graudeinstrucao;
```

3. Combinando com outras funções

Podemos aplicar mais de uma função ao mesmo tempo!

```
SELECT sexo,
       COUNT(*) AS qtde,
       AVG(salario) AS media_salario,
       max(salario) AS total_salario
FROM Movimentacao
GROUP BY sexo;
```

Tente executar os comandos acima. Depois de ver os resultados, com certeza você ficará curioso para obter mais informações. Aproveite para criar selects de acordo com as informações que você deseja.

4. HAVING: Filtrar os Grupos

O HAVING filtra resultados depois do agrupamento, diferente do WHERE, que filtra antes.

```
SELECT sexo,
       COUNT(*) AS qtde,
       AVG(salario) AS media_salario,
       max(salario) AS máximo
FROM Movimentacao
GROUP BY sexo;
```

Para lembrar

Conceito	Quando usar
WHERE	Filtros antes de agrupar
GROUP BY	Quando você quer separar por categoria
HAVING	Filtros depois de agrupar
AVG, SUM...	Quando quiser resumir, calcular ou contar

Você tem acesso à tabela MovCuritiba, que possui as mesmas colunas de Movimentacao. Exercite os comandos Select para responder às suas perguntas.

Vamos para os exercícios básicos?

http://200.17.199.250/siteprototipo/curso/mysql/qrmysql007.html

Subconsultas (Subqueries): Perguntas Dentro de Perguntas

Imagine que você está organizando sua biblioteca e precisa de uma lista de livros, mas só daqueles escritos por autores brasileiros. Como você faria isso? Primeiro, você precisaria saber quem são os autores brasileiros, certo? Depois, usaria essa informação para encontrar os livros.

No MySQL, podemos fazer algo parecido usando **subconsultas**. Uma subconsulta é basicamente uma **consulta SQL que está "aninhada" dentro de outra consulta SQL**. Ela executa primeiro, e o resultado dela é usado pela consulta "de fora".
Pense na subconsulta como uma pergunta auxiliar que você faz ao banco de dados para ajudar a responder a pergunta principal.

Como Funciona na Prática?

Vamos usar as tabelas autores e livros para entender melhor.
Cenário: Queremos encontrar todos os livros escritos por autores nascidos no Brasil.
Passo a passo (mentalmente):

1. Primeiro, precisamos descobrir os id_autor de todos os autores cuja nacionalidade é 'Brasileira'.
2. Depois, usamos esses id_autor para buscar os livros na tabela livros.

Com uma subconsulta, fazemos isso em uma única instrução:
SQL

```
SELECT titulo, ano_publicacao
FROM livros
WHERE id_autor IN (
    SELECT id_autor
    FROM autores
    WHERE nacionalidade = 'Brasileira'
);
```

Vamos entender esse código:

- SELECT titulo, ano_publicacao FROM livros: Esta é a nossa **consulta principal**. Ela está pedindo para selecionar o título e o ano de publicação da tabela livros.
- WHERE id_autor IN (...): Aqui, estamos filtrando os livros. A cláusula IN significa "onde o id_autor está presente na lista de resultados da subconsulta".
- (SELECT id_autor FROM autores WHERE nacionalidade = 'Brasileira'): Esta é a **subconsulta**. Ela será executada primeiro e retornará uma lista de id_autor de todos os autores brasileiros. No nosso caso, ela retornaria os id_autor de Machado de Assis e Clarice Lispector.

Resultado da consulta:

titulo	ano_publicacao
Dom Casmurro	1899
A Hora da Estela	1977
Quincas Borba	1891

Viu? A subconsulta nos ajudou a filtrar os livros de forma eficiente!

UNION e UNION ALL: Combinando Resultados

Às vezes, você precisa unir os resultados de duas ou mais consultas independentes em um único conjunto de resultados. É para isso que usamos os operadores UNION e UNION ALL. Eles são muito úteis quando você tem dados em diferentes partes do seu banco de dados ou quando precisa consolidar informações de diferentes maneiras.

UNION (Remove Duplicatas)

O operador UNION combina os resultados de duas ou mais instruções SELECT. A regra principal é que as consultas que você está unindo devem ter:

- O **mesmo número de colunas**.
- Colunas com **tipos de dados compatíveis** (não precisam ser idênticos, mas devem ser conversíveis, como um VARCHAR e um TEXT).
- A **ordem das colunas deve ser a mesma**.

Quando você usa UNION, o MySQL **remove as linhas duplicadas** do resultado final.

Exemplo Prático

Vamos imaginar que queremos listar todos os nomes de clientes e todos os nomes de autores em uma única lista, sem repetir os nomes que aparecem nas duas tabelas (como "José Saramago" ou "Isabel Allende").

```
SELECT nome FROM Cliente
UNION
SELECT nome FROM autores;
```

Neste exemplo:

- A primeira consulta SELECT nome FROM Cliente retorna todos os nomes da tabela Cliente.
- A segunda consulta SELECT nome FROM autores retorna todos os nomes da tabela autores.
- O UNION combina essas duas listas e remove qualquer nome que apareça em ambas, mostrando cada nome apenas uma vez.

UNION ALL (Mantém Duplicatas)

O operador UNION ALL é muito parecido com UNION, mas com uma diferença crucial: ele **inclui todas as linhas de ambas as consultas, incluindo as duplicadas**. Se a performance for importante e você não se importar com (ou até precisar de) linhas duplicadas, UNION ALL geralmente é mais rápido que UNION porque não precisa fazer o trabalho extra de verificar e remover duplicatas.

Exemplo Prático

Usando o mesmo cenário de antes, se quiséssemos ver todos os nomes de clientes e autores, mas mantendo as duplicatas, poderíamos usar UNION ALL. Isso seria útil se, por exemplo, quiséssemos contar o número total de entradas em ambas as listas, mesmo que um nome se repita.

```sql
SELECT nome FROM Cliente
UNION ALL
SELECT nome FROM autores;
```

Neste caso, nomes como "José Saramago" ou "Isabel Allende" apareceriam duas vezes na lista final, uma vez vindo da tabela Cliente e outra da tabela autores.

Quando usar UNION vs. UNION ALL?

- Use **UNION** quando você precisa de uma lista única de valores, sem repetições. Pense em uma lista de todos os usuários únicos do seu sistema, independentemente de onde eles estejam cadastrados.
- Use **UNION ALL** quando você quer ver todas as ocorrências dos dados, incluindo duplicatas, ou quando a performance é crucial e você sabe que não há duplicatas (ou que elas não importam). Pense em consolidar logs de diferentes sistemas, onde cada entrada é importante.

Vamos para os exercícios?

http://200.17.199.250/siteprototipo/curso/mysql/qrmysql008.html

Consultas Avançadas e Junções (JOINs)

Até agora, aprendemos a manipular dados dentro de uma única tabela. No entanto, o verdadeiro poder dos bancos de dados relacionais reside na capacidade de **combinar dados de múltiplas tabelas** usando **junções (JOINs)**. Além disso, veremos como usar **subconsultas** para realizar operações mais complexas e como **combinar resultados** de diferentes consultas.

Junções (JOINs)

As junções são usadas para combinar linhas de duas ou mais tabelas com base em uma coluna relacionada entre elas. A chave para entender as junções é a **chave estrangeira (FK)** que estabelece o vínculo entre as tabelas.

Vamos usar nossas tabelas `autores` e `livros` para os exemplos. Lembre-se que `livros.id_autor` é uma chave estrangeira que referencia `autores.id_autor`.

INNER JOIN: Recuperação de Registros Correspondentes

O `INNER JOIN` retorna apenas as linhas que têm **correspondência** em ambas as tabelas (onde a condição de junção é verdadeira). É o tipo de JOIN mais comum.

```
SELECT colunas
FROM tabela1
INNER JOIN tabela2 ON tabela1.coluna_comum = tabela2.coluna_comum;
```

Exemplo:
Quero listar o título de cada livro junto com o nome do seu autor.

SQL

```
SELECT
    livros.titulo,
    autores.nome AS nome_do_autor
FROM
    livros
INNER JOIN
    autores ON livros.id_autor = autores.id_autor;
```

Explicação:

- Estamos selecionando o `titulo` da tabela `livros` e o `nome` da tabela `autores`.
- A cláusula `ON` especifica a condição de junção: onde o `id_autor` na tabela `livros` é igual ao `id_autor` na tabela `autores`.
- O resultado mostrará apenas os livros que têm um autor correspondente na tabela `autores`. Se houvesse um livro com um `id_autor` que não existe na tabela `autores`, ele não seria incluído no resultado.

LEFT JOIN (ou LEFT OUTER JOIN): Todos da Esquerda + Correspondentes da Direita

O `LEFT JOIN` retorna **todas as linhas da tabela da esquerda (a primeira tabela listada após `FROM`)** e as linhas correspondentes da tabela da direita. Se não houver correspondência na tabela da direita, as colunas da tabela da direita terão valores `NULL`.
Sintaxe:
SQL

```
SELECT colunas
FROM tabela1
LEFT JOIN tabela2 ON tabela1.coluna_comum = tabela2.coluna_comum;
```

Exemplo:

Quero listar todos os autores e, se eles tiverem livros, os títulos desses livros. Mesmo que um autor não tenha livros, ele deve aparecer.

SQL

```
SELECT
    autores.nome AS nome_do_autor,
    livros.titulo
FROM
    autores -- Esta é a tabela da esquerda
LEFT JOIN
    livros ON autores.id_autor = livros.id_autor;
```

Explicação:

- Todos os autores serão listados.
- Para cada autor, se houver um livro correspondente, o título do livro será exibido.
- Se um autor não tiver livros, o `titulo` para aquele autor será `NULL`.

`RIGHT JOIN` (ou `RIGHT OUTER JOIN`): Todos da Direita + Correspondentes da Esquerda

O `RIGHT JOIN` funciona de forma oposta ao `LEFT JOIN`. Ele retorna **todas as linhas da tabela da direita (a segunda tabela listada após `JOIN`)** e as linhas correspondentes da tabela da esquerda. Se não houver correspondência na tabela da esquerda, as colunas da tabela da esquerda terão valores `NULL`.

Sintaxe:

```
SELECT colunas
FROM tabela1
RIGHT JOIN tabela2 ON tabela1.coluna_comum = tabela2.coluna_comum;
```

Exemplo:

Quero listar todos os livros e, se houver um autor correspondente, o nome do autor. Mesmo que um livro tenha um `id_autor` inválido (que não existe), ele deve aparecer.

```
SELECT
    livros.titulo,
    autores.nome AS nome_do_autor
FROM
    autores -- Esta é a tabela da esquerda (mas o RIGHT JOIN prioriza a direita)
RIGHT JOIN
    livros ON autores.id_autor = livros.id_autor;
```

Explicação:

- Todos os livros serão listados.
- Para cada livro, se houver um autor correspondente, o nome do autor será exibido.
- Se um livro tiver um `id_autor` que não corresponde a nenhum `id_autor` na tabela `autores`, o `nome_do_autor` será `NULL`.

Subconsultas (Subqueries)

Uma **subconsulta** (ou "subquery" ou "consulta aninhada") é uma consulta SQL que é executada dentro de outra consulta SQL. O resultado da subconsulta é usado como entrada para a consulta externa. Elas são úteis para resolver problemas que não podem ser facilmente resolvidos com uma única consulta simples ou JOINs.

Onde usar subconsultas:

- Na cláusula `WHERE` (com operadores `IN`, `EXISTS`, `>`, `<`, etc.).

- Na cláusula FROM (como uma tabela derivada).
- Na cláusula SELECT (como uma coluna escalar).

Exemplo 1: Subconsulta na cláusula WHERE (com IN)

Quero encontrar o nome dos autores que escreveram livros publicados depois de 1900.

```
SELECT nome
FROM autores
WHERE id_autor IN (SELECT id_autor FROM livros WHERE ano_publicacao > 1900);
```

Explicação:

1. A subconsulta (SELECT id_autor FROM livros WHERE ano_publicacao > 1900) é executada primeiro, retornando uma lista de id_autor que escreveram livros após 1900.
2. A consulta externa então usa essa lista para filtrar os autores da tabela autores onde o id_autor está IN essa lista.

Exemplo 2: Subconsulta na cláusula FROM (tabela derivada)

Quero encontrar o número de livros por autor, mas só quero ver o nome do autor.

SQL

```
SELECT
    a.nome,
    livros_contados.total_livros
FROM
    autores AS a
INNER JOIN
    (SELECT id_autor, COUNT(*) AS total_livros FROM livros GROUP BY id_autor) AS livros_contados
ON a.id_autor = livros_contados.id_autor;
```

Explicação:

1. A subconsulta (SELECT id_autor, COUNT(*) AS total_livros FROM livros GROUP BY id_autor) é executada primeiro, criando uma tabela temporária (apelidada de livros_contados) com id_autor e a contagem de livros para cada um.
2. A consulta externa então faz um INNER JOIN entre a tabela autores e esta tabela derivada para obter os nomes dos autores correspondentes.

UNION e UNION ALL: Combinando Resultados de Múltiplas Consultas

Os operadores UNION e UNION ALL são usados para combinar os conjuntos de resultados de duas ou mais instruções SELECT em um único conjunto de resultados.

Regras para usar UNION ou UNION ALL:

- O número de colunas em cada SELECT deve ser o mesmo.
- As colunas correspondentes em cada SELECT devem ter tipos de dados semelhantes (ou compatíveis).
- A ordem das colunas nas instruções SELECT deve ser a mesma.

UNION: Remove Duplicatas

UNION combina os resultados de duas ou mais consultas e **remove as linhas duplicadas** do resultado final.

Sintaxe:

SQL

```
SELECT coluna1, coluna2 FROM tabela1
UNION
SELECT coluna1, coluna2 FROM tabela2;
```

Exemplo:

Suponha que temos uma tabela usuarios_antigos com id, nome, email e queremos combiná-la com a tabela autores (considerando que autores.nome e autores.email se assemelham).

SQL

```sql
-- Criar uma tabela de exemplo para ilustrar
CREATE TABLE usuarios_antigos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE
);

INSERT INTO usuarios_antigos (nome, email) VALUES ('João Silva', 'joao@email.com');
INSERT INTO usuarios_antigos (nome, email) VALUES ('Maria Souza', 'maria@email.com');
INSERT INTO usuarios_antigos (nome, email) VALUES ('Clarice Lispector', 'clarice@literatura.com'); -- Duplicata potencial

-- Combinar nomes e emails de autores e usuários antigos
SELECT nome, email FROM autores
UNION
SELECT nome, email FROM usuarios_antigos;
```

Explicação:

- As linhas `('Clarice Lispector', 'clarice@literatura.com')` apareceriam apenas uma vez no resultado, mesmo que existam em ambas as tabelas, porque `UNION` remove duplicatas.

`UNION ALL`: Inclui Todas as Duplicatas

`UNION ALL` combina os resultados de duas ou mais consultas e **inclui todas as linhas**, mesmo as duplicadas.

Sintaxe:

SQL

```sql
SELECT coluna1, coluna2 FROM tabela1
UNION ALL
SELECT coluna1, coluna2 FROM tabela2;
```

Exemplo:

Usando o mesmo cenário acima:

```sql
SELECT nome, email FROM autores
UNION ALL
SELECT nome, email FROM usuarios_antigos;
```

Explicação:

- As linhas `('Clarice Lispector', 'clarice@literatura.com')` apareceriam duas vezes no resultado (uma vez para cada tabela), porque `UNION ALL` não remove duplicatas.

Dominar JOINs, subconsultas e `UNION`/`UNION ALL` é fundamental para extrair informações complexas e significativas de bancos de dados relacionais. Essas ferramentas permitem que você combine e filtre dados de maneiras que seriam impossíveis com comandos DML básicos. Vamos para os exercícios?

http://200.17.199.250/siteprototipo/curso/mysql/qrmysql006.html

Restrições: Definindo as Regras dos Seus Dados

Restrições são regras que você define para as colunas das suas tabelas para garantir que os dados inseridos nelas sejam válidos e sigam um padrão. Elas são como "filtros" que impedem a entrada de informações incorretas ou inconsistentes.

Vamos explorar as restrições mais comuns:

PRIMARY KEY (Chave Primária)

A PRIMARY KEY é a rainha das restrições! Ela identifica de forma única cada registro em uma tabela. Pense nela como o CPF de uma pessoa: não existem dois CPFs iguais.

Características de uma PRIMARY KEY:

- **Valor Único:** Cada valor na coluna PRIMARY KEY deve ser diferente dos outros.
- **Não Nula (NOT NULL):** Uma PRIMARY KEY nunca pode estar vazia.
- **Identificador Principal:** É a forma mais eficiente de buscar e referenciar um registro específico.

Exemplo na tabela Cliente:

```
CREATE TABLE Cliente (
    id INT AUTO_INCREMENT PRIMARY KEY, -- 'id' é a chave primária
    nome VARCHAR(100) NOT NULL,
    email varchar(200) not null
);
```

Aqui, a coluna id é definida como PRIMARY KEY. Isso significa que cada cliente terá um id exclusivo e que esse id não poderá ser nulo. O AUTO_INCREMENT ainda ajuda, gerando um id automaticamente a cada novo cliente.

FOREIGN KEY (Chave Estrangeira)

Já falamos sobre ela! A FOREIGN KEY garante a integridade referencial, conectando tabelas e mantendo a consistência dos dados entre elas.

Exemplo na tabela livros:

```
CREATE TABLE livros (
    -- ... outras colunas
    autor_id        INT, -- Coluna da chave estrangeira
    FOREIGN KEY (autor_id) REFERENCES autores(id) -- Define a chave estrangeira
);
```

UNIQUE (Único)

A restrição UNIQUE garante que todos os valores em uma coluna sejam únicos, assim como a PRIMARY KEY. A diferença é que você pode ter várias colunas UNIQUE em uma tabela, mas apenas uma PRIMARY KEY. Além disso, uma coluna UNIQUE pode aceitar valores nulos (mas apenas um nulo, pois nulos são considerados únicos entre si).

Exemplo na tabela livros:

```
CREATE TABLE livros (
    -- ... outras colunas
    isbn        VARCHAR(13) UNIQUE -- Garante que cada ISBN seja único
    -- ...
);
```

Aqui, isbn é UNIQUE, o que impede que dois livros tenham o mesmo número de ISBN.

NOT NULL (Não Nulo)

A restrição NOT NULL impede que uma coluna fique sem valor (nula). Se você tentar inserir um registro sem fornecer um valor para uma coluna NOT NULL, o MySQL vai avisar você.

Exemplo na tabela Cliente:

SQL

```
CREATE TABLE Cliente (
    -- ...
    nome VARCHAR(100) NOT NULL, -- 'nome' não pode ser vazio
    email varchar(200) not null -- 'email' não pode ser vazio
);
```

Ambas as colunas nome e email são NOT NULL, garantindo que todo cliente tenha um nome e um e-mail cadastrados.

DEFAULT (Valor Padrão)

A restrição DEFAULT permite que você defina um valor padrão para uma coluna, caso nenhum valor seja especificado quando um novo registro é inserido. Isso é útil para preencher automaticamente informações comuns.

Exemplo na tabela autores:

SQL

```
CREATE TABLE autores (
    -- ...
    nacionalidade VARCHAR(50) DEFAULT 'Desconhecida', -- Se não informar, será 'Desconhecida'
    -- ...
);
```

Se você não informar a nacionalidade de um autor ao adicioná-lo, ela será automaticamente preenchida como 'Desconhecida'.

CHECK (Verificação)

A restrição CHECK permite que você defina uma condição que todos os valores em uma coluna devem satisfazer. Se a condição não for atendida, o MySQL impedirá a inserção ou atualização do dado.

Embora não esteja presente nas tabelas fornecidas, um exemplo seria:

```
CREATE TABLE Produto (
    id INT PRIMARY KEY,
    nome VARCHAR(100),
    preco DECIMAL(10, 2) CHECK (preco > 0) -- Preço deve ser maior que zero
);
```

Nesse caso, a coluna preco só aceitará valores maiores que zero.

Índices: Acelerando Suas Consultas

Imagine um livro sem índice remissivo. Para encontrar um tópico específico, você teria que folhear todas as páginas. Agora, imagine um livro com um índice detalhado: você encontra o tópico em segundos. Os **índices** no MySQL funcionam exatamente assim para suas tabelas!
Um índice é uma estrutura de dados especial que o MySQL cria para acelerar a recuperação de dados de uma tabela. Ele armazena um mapa dos valores de uma ou mais colunas junto com a localização física desses dados.

Como os Índices Funcionam?

Quando você faz uma consulta (SELECT) em uma tabela sem índices adequados, o MySQL pode precisar "varrer" a tabela inteira, linha por linha, para encontrar o que você procura. Isso é chamado de **leitura de tabela completa** (full table scan) e pode ser muito lento em tabelas grandes.
Com um índice, o MySQL pode ir diretamente para as linhas que contêm os dados que você precisa, sem ter que ler a tabela inteira. Isso reduz drasticamente o tempo de resposta da consulta.

Quando Criar Índices?

Criar índices é uma arte e uma ciência. Nem sempre mais índices significam melhor desempenho. Na verdade, muitos índices podem até piorar o desempenho de operações de escrita (inserção, atualização e exclusão), pois cada vez que você modifica um dado, o índice precisa ser atualizado.

Aqui estão algumas diretrizes de quando criar índices:

- **Em colunas usadas frequentemente em cláusulas WHERE:** Se você busca muito por um nome, email, ou competenciamov, considere criar um índice nessas colunas.
- **Em colunas usadas em JOINs:** Se você conecta tabelas usando colunas como autor_id na cláusula JOIN, um índice nessa coluna (que já é uma chave estrangeira e muitas vezes já indexada automaticamente) é crucial.
- **Em colunas usadas para ordenar (ORDER BY):** Se você frequentemente ordena seus resultados por data_nascimento ou salario, um índice pode ajudar.
- **Em colunas com alta cardinalidade (muitos valores únicos):** Colunas como id, email, isbn são bons candidatos, pois cada valor é distinto, tornando o índice muito eficaz. Colunas com poucos valores únicos (ex: sexo - masculino/feminino) geralmente não se beneficiam tanto de índices.
- **Chaves Primárias e Estrangeiras:** O MySQL automaticamente cria índices para suas PRIMARY KEYs. Para FOREIGN KEYs, ele também cria índices na maioria dos casos (mas vale a pena verificar!).

Exemplo de Criação de Índice

Para a tabela Movimentacao, que é bastante grande, podemos identificar colunas que seriam frequentemente usadas em filtros. Por exemplo, se você costuma consultar movimentações por competenciamov (competência da movimentação) ou regiao:
SQL

```
CREATE INDEX idx_competenciamov ON Movimentacao (competenciamov);
CREATE INDEX idx_regiao ON Movimentacao (regiao);
```

Essas duas linhas de comando criam índices nas colunas competenciamov e regiao da tabela Movimentacao. Se você buscar por:
SQL

```
SELECT * FROM Movimentacao WHERE competenciamov = 202401;
```

O MySQL usará o idx_competenciamov para encontrar os dados rapidamente.

Importante: Não crie índices em todas as colunas! Analise suas consultas mais frequentes e identifique onde o desempenho está mais lento. Use a ferramenta EXPLAIN do MySQL (um tópico mais avançado) para entender como suas consultas estão sendo executadas e se os índices estão sendo usados.

Conclusão

As **restrições** são seus guardiões de dados, garantindo que a informação armazenada seja sempre consistente e válida. Elas são a base para um banco de dados confiável. Os **índices**, por sua vez, são seus aceleradores de consulta, transformando buscas lentas em processos rápidos e eficientes.
Dominar esses dois conceitos é fundamental para qualquer pessoa que trabalhe com bancos de dados, mesmo sem ser um especialista em TI. Com eles, você constrói sistemas mais robustos e com melhor desempenho.

Funções de String: Manipulando Textos

As funções de string são perfeitas para trabalhar com dados textuais, como nomes, e-mails ou descrições.

CONCAT(): Juntando Pedaços de Texto

Imagine que você tem o primeiro nome e o sobrenome de um cliente em colunas separadas e quer exibi-los juntos. O CONCAT() faz exatamente isso: ele **concatena** (junta) duas ou mais strings.

Sintaxe: CONCAT(string1, string2, ...)

Exemplo:
Se você tem uma tabela Cliente com as colunas nome e email concatenados:

```
SELECT CONCAT(nome, ' - ', email) AS NomeCompleto
FROM Cliente;
```

Execute a consulta acima para ver o resultado.

LENGTH(): Contando Caracteres

Quer saber quantos caracteres tem um texto? A função LENGTH() retorna o **tamanho** de uma string. Exemplo: Para saber o comprimento do e-mail dos seus clientes:

```
SELECT email, LENGTH(email) AS TamanhoEmail
FROM Cliente;
```

SUBSTRING(): Extraindo Partes de um Texto

Se você precisa pegar apenas uma parte de uma string, o SUBSTRING() é a ferramenta ideal. Você especifica a string, a posição inicial e (opcionalmente) o número de caracteres a serem extraídos.

Sintaxe: SUBSTRING(string, posicao_inicial, [tamanho])

- posicao_inicial: Onde a extração começa (1 é o primeiro caractere).
- tamanho: Quantos caracteres extrair (se omitido, extrai até o final).

Exemplo:Para pegar apenas os três primeiros dígitos do isbn da sua tabela livros:

```
SELECT isbn, SUBSTRING(isbn, 1, 3) AS PrefixoISBN
FROM livros;
```

UPPER() e LOWER(): Mudando o Caso das Letras

Quer padronizar o texto, deixando tudo em maiúsculas ou minúsculas?

- UPPER(): Converte todos os caracteres de uma string para **maiúsculas**.
- LOWER(): Converte todos os caracteres de uma string para **minúsculas**.

Sintaxe:

- UPPER(string)

- LOWER(string)

Exemplo:

Para exibir os nomes dos clientes em maiúsculas e os e-mails em minúsculas:

SQL

```
SELECT UPPER(nome) AS NomeMaiusculo, LOWER(email) AS EmailMinusculo
FROM Cliente;
```

Funções Numéricas: Realizando Cálculos

Para lidar com números, arredondamentos e valores absolutos, as funções numéricas são suas aliadas.

ROUND(): Arredondando Números

A função ROUND() arredonda um número para um certo número de casas decimais.

Sintaxe: ROUND(numero, [casas_decimais])

- casas_decimais: Opcional. Se omitido, arredonda para o número inteiro mais próximo.

Exemplo:

Na tabela Movimentacao, se você quisesse arredondar o salario para duas casas decimais:

```
SELECT salario, ROUND(salario, 2) AS SalarioArredondado
FROM Movimentacao;
```

CEIL() e FLOOR(): Arredondando para Cima ou para Baixo

- CEIL() (Ceiling): Arredonda um número **para cima** para o inteiro mais próximo.
- FLOOR(): Arredonda um número **para baixo** para o inteiro mais próximo.

Sintaxe:

- CEIL(numero)
- FLOOR(numero)

Exemplo: Se o horascontratuais da Movimentacao for um número decimal e você quiser o valor inteiro acima ou abaixo:

SQL

```
SELECT horascontratuais, CEIL(horascontratuais+0.5) AS HorasParaCima,
FLOOR(horascontratuais+0.5) AS HorasParaBaixo
FROM Movimentacao;
```

ABS(): Valor Absoluto

A função ABS() retorna o **valor absoluto** de um número, ou seja, a distância desse número até zero, sempre positivo.

Exemplo: Se você tivesse uma coluna com diferenças de valores que pudessem ser negativas, e você quisesse a diferença independentemente do sinal:

```
SELECT -10, ABS(-10) AS ValorAbsoluto;
```

Funções de Data e Hora: Trabalhando com Tempo

Datas e horas são tipos de dados muito comuns, e o MySQL oferece funções poderosas para manipulá-los.

NOW(), CURDATE(), CURTIME(): A Data e Hora Atuais

Essas funções são autoexplicativas:

- NOW(): Retorna a **data e hora atuais** (ex: 2025-06-26 22:37:53).
- CURDATE(): Retorna apenas a **data atual** (ex: 2025-06-26).
- CURTIME(): Retorna apenas a **hora atual** (ex: 22:37:53).

Exemplo:

```
SELECT NOW() AS DataHoraAtual, CURDATE() AS DataAtual, CURTIME() AS HoraAtual;
```

DATE_FORMAT(): Formatando Datas para Exibição

As datas são armazenadas em um formato padrão (YYYY-MM-DD HH:MM:SS), mas você pode querer exibi-las de outras maneiras (ex: DD/MM/YYYY). A função DATE_FORMAT() permite que você formate uma data de acordo com um padrão que você define.

Sintaxe: DATE_FORMAT(data, formato)

Existem muitos códigos de formato que você pode usar (e pode pesquisar por "MySQL DATE_FORMAT codes" para ver a lista completa), mas alguns comuns são:

- %Y: Ano com quatro dígitos (ex: 2025)
- %m: Mês numérico (01-12)
- %d: Dia do mês numérico (01-31)
- %H: Hora (00-23)
- %i: Minutos (00-59)
- %s: Segundos (00-59)
- %M: Nome completo do mês (ex: June)
- %W: Nome completo do dia da semana (ex: Wednesday)

Exemplo:Na tabela autores, se você quiser formatar a data_nascimento para "DD/MM/YYYY":

```
SELECT nome, DATE_FORMAT(data_nascimento, '%d/%m/%Y') AS DataNascimentoFormatada
FROM autores;
```

DATEDIFF(): Calculando Diferença entre Datas

A função DATEDIFF() calcula a **diferença em dias** entre duas datas.

Sintaxe: DATEDIFF(data1, data2)

- Retorna data1 - data2.

Exemplo: Para saber quantos dias se passaram desde a data_nascimento de um autor até a data atual:

```
SELECT nome, data_nascimento, DATEDIFF(CURDATE(), data_nascimento) AS DiasDesdeNascimento
```

```
FROM autores;
```

Conclusão

As funções do MySQL são ferramentas incrivelmente úteis que te permitem ir muito além da simples recuperação de dados. Elas possibilitam que você manipule textos, realize cálculos e formate datas e horas de acordo com suas necessidades.

Pratique o uso dessas funções com suas próprias tabelas. Comece a pensar em como você pode usá-las para extrair insights ou apresentar seus dados de forma mais clara e útil. A medida que você se familiariza com elas, verá o quão poderosas podem ser para a análise e apresentação dos seus dados!

Data Manipulation Language (DML): O CRUD dos Seus Dados

A DML é a parte do SQL que permite que você **interaja com os dados** que estão dentro das suas tabelas. Pense na DML como as ações que você faz com suas informações: adicionar, atualizar e remover.

Essa tríade de operações é tão fundamental que tem um nome especial no desenvolvimento de software: **CRUD** (Create, Read, Update, Delete). Neste capítulo, vamos focar em três das quatro operações do CRUD:

- **C**reate (Criar) rightarrow INSERT INTO
- **U**pdate (Atualizar) rightarrow UPDATE
- **D**elete (Excluir) rightarrow DELETE FROM

A operação **Read (Ler)**, que é o SELECT, é tão importante e tem tantas variações que já dedicamos alguns capítulos para ela!

Vamos usar as tabelas autores e livros que criamos para praticar esses comandos.

INSERT INTO: Adicionando Novas Informações (C- Create)

O comando INSERT INTO é usado para **adicionar novos registros (linhas)** em uma tabela. É assim que seus dados entram no banco!
Existem duas formas principais de usar o INSERT INTO:

1. Inserir dados em todas as colunas (na ordem correta):

   Se você vai fornecer um valor para todas as colunas da tabela, e na mesma ordem em que elas foram criadas, pode omitir a lista de colunas.

   ```
   INSERT INTO autores VALUES (NULL, 'Machado de Assis', 'Brasileira', '1839-06-21');
   -- Usamos NULL para o 'id' porque ele é AUTO_INCREMENT e o MySQL gerará o valor automaticamente.
   ```

2. Inserir dados em colunas específicas (recomendado):

   Esta é a forma mais segura e recomendada, pois você especifica exatamente quais colunas está preenchendo. Isso é útil se você não tem valores para todas as colunas ou se a ordem das colunas mudar no futuro.

   ```
   -- Inserindo um novo autor
   INSERT INTO autores (nome, nacionalidade, data_nascimento)
   VALUES ('Clarice Lispector', 'Ucraniana', '1920-12-10');

   -- Vamos inserir outro autor para ter mais dados
   INSERT INTO autores (nome, nacionalidade, data_nascimento)
   VALUES ('Gabriel Garcia Marquez', 'Colombiana', '1927-03-06');
   ```

   Agora que temos autores, podemos inserir livros. Lembre-se que autor_id na tabela livros precisa corresponder a um id existente na tabela autores. Para pegar o id de um autor que acabamos de inserir, podemos usar o LAST_INSERT_ID().

```
-- Inserindo um livro para Machado de Assis (supondo que o ID dele seja 1)
INSERT INTO livros (titulo, ano_publicacao, isbn, autor_id)
VALUES ('Dom Casmurro', 1899, '9788582850986', 1);

-- Inserindo um livro para Clarice Lispector (descobrindo o ID dela com SELECT)
-- Primeiro, vamos ver o ID da Clarice (use um SELECT para verificar o ID dela)
SELECT id FROM autores WHERE nome = 'Clarice Lispector'; -- Suponha que o ID retornado seja 2

INSERT INTO livros (titulo, ano_publicacao, isbn, autor_id)
VALUES ('A Hora da Estrela', 1977, '9788532502693', 2);

-- Inserindo mais um livro para o Gabriel Garcia Marquez
SELECT id FROM autores WHERE nome = 'Gabriel Garcia Marquez'; -- Suponha que o ID retornado seja 3

INSERT INTO livros (titulo, ano_publicacao, isbn, autor_id)
VALUES ('Cem Anos de Solidão', 1967, '9788501012576', 3);
```

Dica Importante: Sempre use `SELECT * FROM tabela;` para verificar se seus dados foram inseridos corretamente após cada `INSERT`.

UPDATE: Modificando Informações Existentes (U- Update)

O comando `UPDATE` é usado para **modificar registros (linhas) existentes** em uma tabela. Com ele, você pode alterar o valor de uma ou mais colunas para um ou mais registros.
A sintaxe básica é:

```
UPDATE nome_da_tabela
SET coluna1 = novo_valor1, coluna2 = novo_valor2, ...
WHERE condicao; -- MUITO IMPORTANTE!
```

A cláusula `WHERE` é **essencial** no `UPDATE` (e também no `DELETE`). Ela define quais registros serão afetados. **Se você esquecer o `WHERE`, o `UPDATE` modificará TODOS os registros da tabela!**

Exemplos:

1. Alterar a nacionalidade de um autor:

   Vamos supor que descobrimos que "Clarice Lispector" era, na verdade, naturalizada brasileira e não ucraniana para fins de registro.

   ```
   UPDATE autores
   SET nacionalidade = 'Brasileira'
   WHERE nome = 'Clarice Lispector';
   ```
2. Atualizar o ano de publicação de um livro:

   Imagine que o isbn '9788582850986' foi registrado com o ano errado.

   ```
   UPDATE livros
   SET ano_publicacao = 1899
   WHERE isbn = '9788582850986';
   ```

3. Atualizar múltiplas colunas:

   Vamos corrigir o ano de publicação e adicionar uma nova informação para "Cem Anos de Solidão".

```
UPDATE livros
SET ano_publicacao = 1967, titulo = 'Cem Anos de Solidão - Edição Comemorativa'
WHERE isbn = '9788501012576';
```

Sempre use WHERE com UPDATE! É a sua segurança para não bagunçar o banco de dados inteiro. Teste a condição WHERE primeiro com um SELECT para ter certeza de que ela seleciona os registros corretos antes de executar o UPDATE.

DELETE FROM: Removendo Informações (D- Delete)

O comando DELETE FROM é usado para **remover registros (linhas)** de uma tabela.
A sintaxe básica é:

```
DELETE FROM nome_da_tabela
WHERE condicao; -- MUITO IMPORTANTE!
```

Assim como no UPDATE, a cláusula WHERE é **fundamental** no DELETE FROM. Ela especifica quais registros serão excluídos. **Se você esquecer o WHERE, o DELETE FROM removerá TODOS os registros da tabela!**

Exemplos:

1. Excluir um livro específico:

   Vamos remover o livro "A Hora da Estrela" do nosso banco.

   SQL

```
DELETE FROM livros
WHERE titulo = 'A Hora da Estrela';
```

2. Excluir um autor (e seus livros, se permitido pelas restrições):

   Aqui, a FOREIGN KEY entra em ação. Se você tentar excluir um autor que ainda tem livros associados a ele, o MySQL pode bloquear a exclusão (dependendo de como a FOREIGN KEY foi configurada, por padrão ele impede para manter a integridade). Primeiro, você teria que excluir os livros do autor, ou configurar a FK com ON DELETE CASCADE (que não é o padrão e deve ser usado com cautela).

   Vamos remover "Gabriel Garcia Marquez" e seu livro. Primeiro, removemos o livro.

   SQL

```
DELETE FROM livros
WHERE autor_id = (SELECT id FROM autores WHERE nome = 'Gabriel Garcia Marquez');

-- Agora podemos remover o autor
DELETE FROM autores
```

```
WHERE nome = 'Gabriel Garcia Marquez';
```

Atenção Redobrada: O comando DELETE FROM é irreversível (a menos que você esteja usando transações com ROLLBACK, como aprendemos em outro capítulo). **Sempre use WHERE com DELETE FROM e tenha certeza da sua condição!** Um DELETE FROM tabela; sem WHERE apaga todos os dados da tabela, o que pode ser um desastre.

Você se sente mais confiante para adicionar, modificar e remover dados das suas tabelas agora? Vamos para os exercícios?

http://200.17.199.250/siteprototipo/curso/mysql/qrmysql009.html

Stored Procedures: Automatizando Tarefas no MySQL

Chegamos a um tópico que pode parecer um pouco mais técnico, mas é incrivelmente útil: as **Stored Procedures** (ou "Procedimentos Armazenados"). Pense nelas como pequenas "receitas" de SQL que você guarda dentro do seu banco de dados. Em vez de escrever e reescrever a mesma sequência de comandos SQL várias vezes, você os empacota em uma stored procedure e simplesmente a "chama" quando precisar.

O Que São Stored Procedures?

Uma **Stored Procedure** é um conjunto de uma ou mais instruções SQL (como `SELECT`, `INSERT`, `UPDATE`, `DELETE`) que são armazenadas, compiladas e executadas diretamente no servidor do banco de dados MySQL. É como ter um programa mini-SQL guardado e pronto para usar.

Imagine que você precisa frequentemente atualizar o salário de funcionários, ou registrar uma nova venda, ou gerar um relatório complexo. Em vez de digitar os mesmos comandos SQL toda vez, você cria uma stored procedure para cada uma dessas tarefas. Da próxima vez, é só chamar o nome da sua "receita".

Vantagens das Stored Procedures

Por que se dar ao trabalho de criar stored procedures? Elas trazem benefícios significativos:

* **Reutilização de Código:** A maior vantagem! Escreva o código uma vez, armazene-o e execute-o quantas vezes quiser. Isso evita repetição, torna seu banco de dados mais organizado e facilita a manutenção. Se precisar mudar a lógica, mude em um só lugar.
* **Melhora no Desempenho:** Como as stored procedures são armazenadas e pré-compiladas no servidor, elas podem ser executadas mais rapidamente do que enviar a mesma sequência de comandos SQL repetidamente do seu aplicativo. O servidor já sabe o que fazer.
* **Segurança (Controle de Acesso):** Você pode conceder permissão para um usuário **executar** uma stored procedure, sem dar a ele permissão direta para acessar ou modificar as tabelas subjacentes. Isso é ótimo para controlar quem pode fazer o quê no seu banco de dados, aumentando a segurança.
* **Redução do Tráfego de Rede:** Em vez de enviar várias instruções SQL pela rede, você envia apenas uma chamada para a stored procedure, o que economiza tempo e largura de banda.

Sintaxe Básica: Criando e Chamando Stored Procedures

Vamos ver como se cria e executa uma stored procedure.

DELIMITER: Um Detalhe Importante

Antes de criar uma stored procedure, você verá um comando `DELIMITER`. O que é isso? Normalmente, o MySQL entende que um comando SQL termina com um ponto e vírgula (`;`). No entanto, dentro de uma stored procedure, pode haver vários pontos e vírgula. Para que o MySQL não pense que cada linha com `;` é o fim da sua procedure, usamos `DELIMITER` para temporariamente mudar o caractere que marca o fim de um comando. Geralmente, usamos `//` ou `$$`. Depois de criar a procedure, você deve voltar o `DELIMITER` para `;`.

CREATE PROCEDURE: Criando Sua "Receita"

A sintaxe básica para criar uma stored procedure é:

```
DELIMITER //
```

```
CREATE PROCEDURE olamundo()
BEGIN
    -- Suas instruções SQL aqui
    SELECT 'Olá, mundo!';
END //
DELIMITER ; -- Volta o delimitador para o padrão
```

- CREATE PROCEDURE olamundo(): Define o nome da sua stored procedure. Os parênteses () são para parâmetros (que veremos a seguir).
- BEGIN e END: Marcam o início e o fim do bloco de instruções da stored procedure.
- -- Suas instruções SQL aqui: Onde você coloca o código que a procedure vai executar.

CALL: Executando Sua "Receita"

Para rodar uma stored procedure que você criou, basta usar o comando CALL:

```
CALL olamundo();
```

Exemplo Prático 1: Uma Stored Procedure Simples

Vamos criar uma stored procedure que apenas exibe uma mensagem.

```
DELIMITER //
CREATE PROCEDURE livrosSel()
BEGIN
   -- Suas instruções SQL aqui
   SELECT titulo, ano_publicacao
   from Livros
   order by 1;
END //
DELIMITER ; -- Volta o delimitador para o padrão

-- Para executar:
CALL livrosSel();
```

Tente criar sua stored procedure no MySQL Workbench e veja a mesma no ‘Schemas’

Parâmetros: Tornando Suas Procedures Flexíveis

As stored procedures ficam muito mais poderosas quando você as torna flexíveis, permitindo que elas aceitem **parâmetros**. Parâmetros são como "ingredientes" que você fornece à sua "receita" para que ela possa agir de forma diferente dependendo do que você passa para ela.

Existem três tipos de parâmetros:

- IN: **Parâmetros de Entrada**. Os mais comuns. Você fornece um valor à procedure, e ela o utiliza em suas operações. A procedure não modifica o valor do parâmetro.
- OUT: **Parâmetros de Saída**. A procedure calcula um valor e o "devolve" para você através desse parâmetro.
- INOUT: **Parâmetros de Entrada e Saída**. Você fornece um valor para a procedure, ela pode modificá-lo e depois "devolver" o novo valor.

Sintaxe de Parâmetros:

```
CREATE PROCEDURE nome_da_procedure(
   IN parametro_entrada TIPO_DADO,
```

```
    OUT parametro_saida TIPO_DADO,
    INOUT parametro_entrada_saida TIPO_DADO
)
BEGIN
    -- ...
END //
```

Exemplo Prático 2: Inserindo um Cliente com Parâmetros IN

Vamos criar uma stored procedure para adicionar um novo cliente na tabela Cliente:

SQL

```
DELIMITER //

CREATE PROCEDURE AdicionarCliente(
    IN p_nome VARCHAR(100),
    IN p_email VARCHAR(200)
)
BEGIN
    INSERT INTO Cliente (nome, email) VALUES (p_nome, p_email);
END //

DELIMITER ;

-- Para executar e adicionar um cliente:
CALL AdicionarCliente('Alice Wonderland', 'alice@wonderland.com');
```

Agora, você pode adicionar clientes de forma padronizada, passando apenas o nome e o e-mail.

Variáveis: Armazenando Informações Temporariamente

Dentro de uma stored procedure, você pode usar **variáveis** para armazenar valores temporariamente. Isso é útil para guardar resultados de cálculos, pedaços de texto ou qualquer informação que você precise usar em diferentes partes da sua procedure.

Declaração de Variáveis

Você declara uma variável usando DECLARE e um nome que começa com @ (variáveis de sessão) ou sem @ (variáveis locais da procedure). Para variáveis locais da procedure, você define o tipo de dado e, opcionalmente, um valor padrão.

Sintaxe: DECLARE nome_variavel TIPO_DADO [DEFAULT valor_padrao];

Uso de Variáveis

Você atribui valores a variáveis usando SET ou SELECT ... INTO.

Exemplo:

SQL

```
DELIMITER //

CREATE PROCEDURE ContarLivrosAutor(
    IN p_autor_id INT,
    OUT p_total_livros INT
)
BEGIN
    DECLARE v_nome_autor VARCHAR(100); -- Declara uma variável para o nome do autor
```

```
    -- Armazena o nome do autor na variável
    SELECT nome INTO v_nome_autor FROM autores WHERE id = p_autor_id;

    -- Conta os livros e armazena na variável de saída
    SELECT COUNT(*) INTO p_total_livros
    FROM livros
    WHERE autor_id = p_autor_id;

    -- Exemplo de uso da variável (opcional, apenas para demonstração)
    SELECT CONCAT('O autor ', v_nome_autor, ' tem ', p_total_livros, ' livros.');

END //

DELIMITER ;

-- Para executar e ver o resultado (observe a variável de sessão para o OUT):
SET @total = 0; -- Declara a variável de sessão para receber o OUT
CALL ContarLivrosAutor(1, @total); -- Supondo que 1 seja um ID de autor válido
SELECT @total AS TotalDeLivros; -- Mostra o valor da variável de saída
```

Neste exemplo, `v_nome_autor` é uma variável local dentro da procedure, e `@total` é uma variável de sessão que usamos para capturar o valor do parâmetro `OUT`.

Controle de Fluxo: Tomando Decisões e Repetindo Ações

Stored procedures podem ser ainda mais inteligentes com **controle de fluxo**, permitindo que elas tomem decisões ou repitam ações.

`IF`, `ELSEIF`, `ELSE`: Instruções Condicionais

Permitem que sua procedure execute diferentes blocos de código dependendo de uma condição ser verdadeira ou falsa.

Sintaxe:

```
IF condicao THEN
    -- Código se a condição for verdadeira
ELSEIF outra_condicao THEN
    -- Código se a outra condição for verdadeira
ELSE
    -- Código se nenhuma das condições anteriores for verdadeira
END IF;
```

Exemplo: Vamos simular um aumento de salário na tabela Movimentacao com base em uma condição:

```
DELIMITER //

CREATE PROCEDURE AjustarSalario(
    IN p_competencia INT,
    IN p_secao VARCHAR(10)
)
BEGIN
    DECLARE v_salario_medio DECIMAL(10,2);

    -- Calcula o salário médio para a seção na competência
    SELECT AVG(salario)
    INTO v_salario_medio
    FROM Movimentacao
```

```
    WHERE competenciamov = p_competencia AND secao = p_secao;

    IF v_salario_medio IS NULL THEN
        SELECT 'Nenhum registro encontrado para a seção e competência.';
    ELSEIF v_salario_medio < 2000.00 THEN
        UPDATE Movimentacao
        SET salario = salario * 1.05 -- Aumenta 5%
        WHERE competenciamov = p_competencia AND secao = p_secao;
        SELECT CONCAT('Salário médio baixo (', v_salario_medio, '). Salários ajustados em 5% para a seção ', p_secao, ' na competência ', p_competencia, '.');
    ELSE
        SELECT CONCAT('Salário médio adequado (', v_salario_medio, '). Nenhum ajuste necessário para a seção ', p_secao, ' na competência ', p_competencia, '.');
    END IF;

END //

DELIMITER ;

-- Exemplo de uso:
-- CALL AjustarSalario(202401, 'ADM');
```

LOOP, WHILE, REPEAT: Estruturas de Repetição

Permitem que sua procedure execute um bloco de código repetidamente. São mais avançadas e geralmente usadas para processamento de dados linha a linha ou iterações complexas.

- WHILE condicao DO ... END WHILE;: Repete enquanto a condição for verdadeira.
- LOOP ... END LOOP;: Repete indefinidamente até um LEAVE explícito.
- REPEAT ... UNTIL condicao END REPEAT;: Repete até a condição ser verdadeira (executa pelo menos uma vez).

Para um curso básico, o IF é o mais relevante para começar. Estruturas de repetição são para cenários mais complexos e exigem um entendimento mais aprofundado do fluxo de dados.

Gerenciamento de Stored Procedures

Depois de criar suas stored procedures, você pode gerenciá-las.

SHOW PROCEDURE STATUS: Listando Suas Procedures

Para ver quais stored procedures estão armazenadas no seu banco de dados, use:

```
SHOW PROCEDURE STATUS WHERE Db = 'nome_do_seu_banco_de_dados';
-- Ou apenas:
SHOW PROCEDURE STATUS; -- Para listar todas as procedures em todos os bancos que você tem acesso
```

Isso mostrará uma lista de suas procedures com informações como nome, banco de dados, criador e data de criação.

DROP PROCEDURE: Excluindo Stored Procedures

Se você não precisa mais de uma stored procedure, pode excluí-la:

```
DROP PROCEDURE IF EXISTS nome_da_procedure;
```

O IF EXISTS é uma boa prática para evitar um erro caso a procedure já não exista.

Conclusão

As **Stored Procedures** são ferramentas poderosas que transformam seu banco de dados de um mero armazenador de informações em um centro de processamento de dados. Elas permitem automatizar tarefas, melhorar o desempenho, e aumentar a segurança e a organização do seu código SQL.

Comece com as mais simples, usando parâmetros IN e o controle de fluxo IF. À medida que você se sentir mais confortável, poderá explorar a complexidade de OUT e outras estruturas de controle de fluxo.

Pense em quais operações repetitivas você faz no seu dia a dia com o banco de dados. Será que uma stored procedure poderia te ajudar a automatizá-las? Como exercícios:

Par cada tabela: Cliente, Livros e Autores, crie:

- stored procedure para atualizar um registro
- stored procedures para apagar um registro

Gerência de Transações: COMMIT e ROLLBACK

Seja bem-vindo a um dos conceitos mais importantes em bancos de dados: as **Transações**. Imagine que você está fazendo uma série de mudanças em seu banco de dados, como adicionar um novo livro e, em seguida, registrar seu autor. O que acontece se algo der errado no meio do caminho? As transações garantem que suas informações fiquem sempre corretas e seguras.

O que são Transações?

No mundo do MySQL, uma **transação** é como uma "operação gigante" que agrupa várias ações menores (como adicionar, atualizar ou deletar dados) em uma única unidade de trabalho. Pense nela como um pacto: ou todas as ações dentro da transação são concluídas com sucesso, ou nenhuma delas é.

Para garantir que seus dados estejam sempre seguros e consistentes, as transações seguem quatro princípios fundamentais, conhecidos pela sigla **ACID**:

- **A**tomicidade: Tudo ou nada! Se uma transação começa, ela deve ser totalmente concluída. Se alguma parte dela falhar, todas as outras partes são desfeitas, como se nunca tivessem acontecido.
- **C**onsistência: A transação leva o banco de dados de um estado válido para outro estado válido. Isso significa que ela nunca vai deixar seus dados em um estado bagunçado ou contraditório.
- **I**solamento: Transações simultâneas não devem interferir umas nas outras. É como se cada transação tivesse sua própria "bolha" de trabalho até ser finalizada.
- **D**urabilidade: Uma vez que uma transação é concluída e confirmada, as mudanças são permanentes e não serão perdidas, mesmo que o sistema caia ou a energia acabe.

Esses quatro pilares garantem a **integridade dos dados**, ou seja, que suas informações no banco de dados sejam sempre precisas, confiáveis e estejam em ordem.

Controle de Transações

Agora que você entende o que são transações, vamos ver como controlá-las no MySQL. É como ter um "liga e desliga" para suas operações no banco de dados.

START TRANSACTION ou BEGIN

Para iniciar uma transação, você usa o comando START TRANSACTION ou, de forma mais curta, BEGIN. Isso avisa ao MySQL que as próximas instruções SQL devem ser tratadas como parte de uma única transação, aguardando uma decisão final.

Exemplo: Imagine que queremos adicionar um novo autor e um livro. Queremos que ambas as operações aconteçam ou que nenhuma delas aconteça.

```
START TRANSACTION;

-- Adicionando um novo autor
INSERT INTO autores (nome, nacionalidade, data_nascimento) VALUES ('Maria Silva', 'Brasileira', '1980-05-15');

-- Vamos supor que queremos pegar o ID do autor recém-inserido para o livro
SELECT @ultimo_autor_id := LAST_INSERT_ID();

-- Adicionando um novo livro para a Maria Silva
```

```
INSERT INTO livros (titulo, ano_publicacao, isbn, autor_id) VALUES ('Aventuras no Campo', 2023, '9788575225010', @ultimo_autor_id);
```

COMMIT

O comando COMMIT é a sua "luz verde". Ele **confirma** todas as alterações que você fez dentro da transação e as salva permanentemente no banco de dados. Uma vez que você executa um COMMIT, as alterações se tornam visíveis para outras pessoas (ou outras transações) que estão acessando o banco de dados e são garantidas para não serem perdidas, mesmo se o servidor for reiniciado.

Exemplo: Continuando o exemplo anterior, se tudo deu certo ao adicionar o autor e o livro, podemos "comitar" essas mudanças.

Após o COMMIT, tanto o novo autor quanto o novo livro estarão definitivamente no seu banco de dados.

ROLLBACK

O ROLLBACK é o "botão de emergência". Ele **desfaz** todas as alterações que foram feitas dentro de uma transação, revertendo o banco de dados para o estado em que estava antes de você iniciar a transação com START TRANSACTION ou BEGIN.

Quando usar ROLLBACK?

- **Erro:** Se algo der errado durante suas operações, como uma tentativa de inserir um ISBN que já existe.
- **Inconsistência:** Se você perceber que as mudanças que está fazendo vão deixar seus dados desorganizados ou incorretos.
- **Operação incompleta:** Se, por algum motivo, você não consegue finalizar todas as ações planejadas dentro da transação.

Exemplo: Imagine que, ao tentar adicionar o livro, você insere um ISBN duplicado, o que causaria um erro. Em vez de deixar o banco de dados em um estado inconsistente, você pode reverter tudo.

```
START TRANSACTION;

-- Adicionando um novo autor
INSERT INTO autores (nome, nacionalidade, data_nascimento) VALUES ('João Souza', 'Português', '1975-11-20');

-- Vamos supor que queremos pegar o ID do autor recém-inserido para o livro
SELECT @ultimo_autor_id := LAST_INSERT_ID();

-- Tentativa de adicionar um livro com um ISBN que já existe (isso causaria um erro na vida real)
INSERT INTO livros (titulo, ano_publicacao, isbn, autor_id) VALUES ('Viagem ao Desconhecido', 2024, '9788575225010', @ultimo_autor_id);

-- Se algo deu errado, desfazemos tudo!
ROLLBACK;
```

Após o ROLLBACK, nem o "João Souza" nem o "Viagem ao Desconhecido" serão adicionados ao banco de dados. É como se a transação nunca tivesse acontecido.

AutoCommit

Por padrão, o MySQL opera em modo AUTOCOMMIT. O que isso significa? Significa que cada instrução SQL que você executa (como um INSERT, UPDATE ou DELETE) é automaticamente tratada como sua própria transação e é **confirmada imediatamente** após sua execução.

Exemplo: Se você simplesmente executar:

```
INSERT INTO autores (nome) VALUES ('Ana Lima');
```

O MySQL automaticamente executa um COMMIT após essa linha, e "Ana Lima" é gravada no banco de dados imediatamente.

Para ter controle manual sobre as transações (o que é muito importante para garantir a integridade dos dados em operações complexas), você pode desabilitar o AUTOCOMMIT e, em seguida, habilitá-lo novamente quando não precisar mais de controle manual.

- **Desabilitar AUTOCOMMIT:**
  SQL

  ```
  SET autocommit = 0;
  ```
  Com o autocommit desabilitado, você **sempre** precisará usar COMMIT para salvar suas mudanças ou ROLLBACK para desfazê-las.
- **Habilitar AUTOCOMMIT:**
  SQL

  ```
  SET autocommit = 1;
  ```
  Isso retorna o MySQL ao seu comportamento padrão, onde cada instrução é confirmada automaticamente.

Importante: Lembre-se de que ao desabilitar o AUTOCOMMIT, suas mudanças só serão salvas se você usar COMMIT. Se você fechar sua conexão com o banco de dados sem um COMMIT, todas as alterações pendentes serão perdidas!

Pontos de Salvamento (SAVEPOINT)

Os SAVEPOINTs (ou pontos de salvamento) são uma ferramenta um pouco mais avançada, mas muito útil em transações longas e complexas. Imagine que você tem uma transação com muitas etapas. Se algo der errado na etapa 5, você não precisa desfazer *todas* as etapas. Com um SAVEPOINT, você pode "voltar" apenas até um ponto específico dentro da transação, sem desfazer tudo.

SAVEPOINT nome_do_ponto

Cria um ponto de salvamento com um nome que você escolhe.

Exemplo:

```
START TRANSACTION;

INSERT INTO autores (nome) VALUES ('Carlos Dantas');
SAVEPOINT antes_livro_1; -- Criei um ponto de salvamento aqui

INSERT INTO livros (titulo, autor_id) VALUES ('A Cidade Secreta', LAST_INSERT_ID());
SAVEPOINT antes_livro_2; -- Criei outro ponto de salvamento aqui

INSERT INTO livros (titulo, autor_id) VALUES ('O Mistério da Floresta', LAST_INSERT_ID());

ROLLBACK TO nome_do_ponto
```

Reverte a transação apenas até o ponto de salvamento especificado. As operações *antes* do SAVEPOINT permanecem intactas.

Exemplo: Se, no exemplo anterior, o INSERT do "O Mistério da Floresta" deu algum problema e você quer desfazer apenas essa última inserção, mas manter "A Cidade Secreta":

SQL

```
ROLLBACK TO antes_livro_2;
```

```
-- Agora, "O Mistério da Floresta" foi desfeito, mas "A Cidade Secreta" e "Carlos Dantas" continuam na transação.
```

```
RELEASE SAVEPOINT nome_do_ponto
```

Remove um ponto de salvamento. Isso não desfaz nenhuma alteração, apenas significa que você não pode mais usar aquele SAVEPOINT específico para um ROLLBACK. As alterações feitas *após* o ponto de salvamento permanecem como parte da transação.

Exemplo:

SQL

```
RELEASE SAVEPOINT antes_livro_1;
-- O ponto de salvamento 'antes_livro_1' não pode mais ser usado, mas as alterações feitas até ali continuam na transação.
```

Entender e usar a gerência de transações é um passo crucial para se tornar um usuário mais eficiente e seguro de bancos de dados. Com COMMIT e ROLLBACK, você tem o poder de controlar suas operações e garantir que seus dados estejam sempre impecáveis!

Exercício: crie uma stored procedure que faça a inserção de um livro e um autor dentro de uma transação.

Modelagem de Banco de Dados: O Mapa da Mina dos Seus Dados

Bem-vindo a um dos passos mais cruciais na criação de um banco de dados: a **modelagem**. Pense na modelagem de banco de dados como o projeto arquitetônico de uma casa. Antes de você colocar um único tijolo, precisa saber quantos quartos, onde fica a cozinha, e como tudo se conecta. Com dados, é a mesma coisa: antes de criar as tabelas no MySQL, precisamos entender quais informações teremos e como elas se relacionam.

Introdução ao Conceito de Modelagem

A **modelagem de banco de dados** é o processo de projetar a estrutura de um banco de dados. É como desenhar um mapa detalhado de todas as informações que você quer guardar e como elas se organizam. O objetivo é criar um design eficiente, que evite dados repetidos (redundância) e garanta que suas informações sejam consistentes e fáceis de acessar.

Uma boa modelagem é a base para um banco de dados que funcione bem, seja fácil de manter e cresça junto com suas necessidades. Sem ela, é como tentar construir uma casa sem planta: o resultado provavelmente será uma bagunça!

Entidades, Atributos e Relacionamentos: Os Pilares da Modelagem

Para começar a modelar, precisamos entender três conceitos-chave:

Entidades

Uma **entidade** representa um "objeto" ou "coisa" do mundo real que você quer armazenar no seu banco de dados. Pense em algo que tenha características e que você precise guardar informações sobre ele.

Exemplos de entidades:

- **Livros** (você quer guardar informações sobre livros)
- **Autores** (você quer guardar informações sobre autores)
- **Alunos** (você quer guardar informações sobre alunos)
- **Produtos** (você quer guardar informações sobre produtos)

No seu banco de dados, cada entidade geralmente se tornará uma **tabela**.

Atributos

Um **atributo** é uma característica ou propriedade de uma entidade. São os detalhes que descrevem cada entidade.

Exemplos de atributos:

- Para a entidade **Livros**:
  - titulo
  - ano_publicacao
  - isbn
- Para a entidade **Autores**:
  - nome
  - nacionalidade
  - data_nascimento

No seu banco de dados, cada atributo geralmente se tornará uma **coluna** dentro da tabela correspondente à entidade.

Relacionamentos

Um **relacionamento** descreve como duas ou mais entidades se conectam entre si. É a forma como as "coisas" interagem no seu mundo de dados.

Tipos comuns de relacionamentos:

- **Um para Um (1:1):** Uma ocorrência da Entidade A se relaciona com uma *única* ocorrência da Entidade B, e vice-versa. (Exemplo: Um país tem uma única capital, e uma capital pertence a um único país).
- **Um para Muitos (1:N ou 1:M):** Uma ocorrência da Entidade A se relaciona com *várias* ocorrências da Entidade B, mas uma ocorrência da Entidade B se relaciona com *apenas uma* ocorrência da Entidade A. (Exemplo: Um autor escreve vários livros, mas um livro é escrito por apenas um autor. *No nosso caso de estudo, consideramos essa simplicidade para começar.*)
- **Muitos para Muitos (N:N ou M:M):** Várias ocorrências da Entidade A se relacionam com *várias* ocorrências da Entidade B, e vice-versa. (Exemplo: Um aluno pode cursar várias disciplinas, e uma disciplina pode ter vários alunos).

Para o nosso exemplo de **livros** e **autores**, temos um relacionamento **Um para Muitos**: um autor pode escrever vários livros, mas (por simplificação para este curso) cada livro tem apenas um autor.

Diagramas ER (Entidade-Relacionamento) Simples

Uma das melhores ferramentas para visualizar seu modelo de dados é o **Diagrama Entidade-Relacionamento (DER)**. Ele é como um fluxograma que usa símbolos padronizados para representar entidades, atributos e seus relacionamentos.

Vamos desenhar um DER simples para nossas entidades autores e livros:

- **Retângulos** representam **entidades**.
- **Ovais** (ou listas dentro dos retângulos) representam **atributos**.
- **Linhas** conectam entidades e representam **relacionamentos**. Símbolos nas pontas das linhas indicam o tipo de relacionamento (1 para 1, 1 para muitos, etc.).

Aqui está como nosso DER se pareceria (imagine os símbolos):

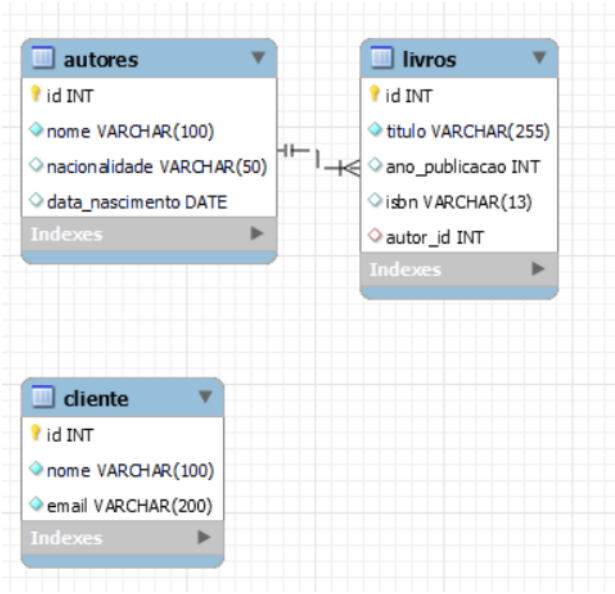


Figura: DER obtido com a engenharia reversa do MySQL Workbench.

Observação: Em um DER real, as linhas de relacionamento teriam símbolos específicos (como "pés de galinha" para o lado "muitos") para indicar o tipo de relacionamento, mas para este curso básico, o importante é entender a ideia.

Como traduzir um modelo para um banco de dados?

Depois de ter seu DER pronto, a "mágica" acontece: você o traduz para a linguagem do MySQL. É aí que as entidades viram tabelas e os atributos viram colunas.

1. **Cada entidade se torna uma tabela:**
   - A entidade **Autores** vira a tabela autores.
   - A entidade **Livros** vira a tabela livros.
2. **Cada atributo se torna uma coluna na tabela correspondente:**
   - Na tabela autores, teremos colunas como id, nome, nacionalidade, data_nascimento.
   - Na tabela livros, teremos colunas como id, titulo, ano_publicacao, isbn.
3. **Os relacionamentos são representados por chaves estrangeiras:**
   - No nosso relacionamento "Um para Muitos" entre autores e livros, o MySQL precisa de uma forma de saber qual autor escreveu qual livro. Para isso, criamos uma **chave estrangeira** (FOREIGN KEY).
   - A coluna autor_id na tabela livros é a chave estrangeira. Ela "aponta" para o id (chave primária) na tabela autores. É assim que o MySQL sabe que um livro pertence a um autor específico.

Vejamos as tabelas que você já tem e como elas refletem essa modelagem:

```
CREATE TABLE autores (
   id              INT AUTO_INCREMENT PRIMARY KEY, -- 'id' é a chave primária, identifica cada autor de
forma única
   nome            VARCHAR(100) NOT NULL,
   nacionalidade   VARCHAR(50) DEFAULT 'Desconhecida',
   data_nascimento DATE null
);

CREATE TABLE livros (
   id              INT AUTO_INCREMENT PRIMARY KEY, -- 'id' é a chave primária da tabela livros
   titulo          VARCHAR(255) NOT NULL,
   ano_publicacao  INT,
   isbn            VARCHAR(13) UNIQUE, -- 'isbn' deve ser único para cada livro
   autor_id        INT, -- Esta é a CHAVE ESTRANGEIRA! Ela armazena o 'id' do autor
   FOREIGN KEY (autor_id) REFERENCES autores(id) -- Isso define o relacionamento: 'autor_id' em
livros referencia 'id' em autores
);
```

Perceba como a coluna autor_id na tabela livros é fundamental para conectar um livro ao seu respectivo autor na tabela autores. Sem ela, o MySQL não saberia qual autor está associado a cada livro!

A modelagem de dados é a etapa de planejamento que garante que seu banco de dados seja uma ferramenta poderosa e organizada, em vez de um emaranhado de informações. Ao entender entidades, atributos e relacionamentos, e como eles se traduzem em tabelas e colunas, você está no caminho certo para construir sistemas de dados eficientes!

Conclusão: Sua Jornada no Mundo do MySQL Começou!

Você chegou ao fim do nosso curso básico de MySQL e teve uma visão geral dos conceitos fundamentais que são o alicerce de qualquer aplicação que lida com dados. Pense em tudo que você aprendeu, desde as primeiras consultas até às stored procedures.

Você agora possui uma base sólida para interagir com bancos de dados, criar suas próprias estruturas, inserir, consultar e manipular informações de forma eficiente e segura. Agora, você pode começar a trabalhar em um projeto como aprendiz, observando o trabalho e desenvolvendo seus projetos. Saiba que o MySQL é uma ferramenta poderosa, e este curso te deu as chaves para começar a desenvolver o seu potencial.

Lembre-se que o aprendizado no mundo da tecnologia é contínuo. Continue praticando, experimentando e explorando as vastas possibilidades que o MySQL oferece. As habilidades que você adquiriu são extremamente valiosas e aplicáveis em diversas áreas, desde a gestão de pequenos projetos pessoais até o trabalho com grandes volumes de dados como as bases de dados públicas.

Sua jornada como "aventureiro dos dados" está apenas começando! Qual será o próximo passo na sua exploração do universo dos bancos de dados? Que tal ser voluntário em um de nossos projetos de extensão?

Sucessos e até a próxima!!

Celso Yoshikazu Ishida

Tabelas de exemplos

```
CREATE TABLE Movimentacao (
    competenciamov INT,
    regiao INT,
    uf INT,
    municipio INT,
    secao VARCHAR(10),
    subclasse INT,
    saldomovimentacao INT,
    cbo2002ocupacao INT,
    categoria INT,
    graudeinstrucao INT,
    idade INT null,
    horascontratuais DECIMAL(5,2),
    racacor INT,
    sexo INT,
    tipoempregador INT,
    tipoestabelecimento INT,
    tipomovimentacao INT,
    tipodedeficiencia INT,
    indtrabintermitente INT,
    indtrabparcial INT,
    salario DECIMAL(10,2),
    tamestabjan INT,
    indicadoraprendiz INT,
    origemdainformacao INT,
    competenciadec INT,
    indicadordeforadoprazo INT,
    unidadesalariocodigo INT,
    valorsalariofixo DECIMAL(10,2)
);
```

Descrição de algumas colunas, estas colunas contêm apenas números. Informações completas no Layout dos Microdados Não-Identificados do Novo CAGED - Base de Movimentações. Uma cópia pode ser encontrada em: http://200.17.199.250/siteprototipo/curso/mysql/qrmysql011.html

Variável	Descrição
Competenciamov	Ano e Mês da movimentação
município	Código do Município
saldomovimentacao	Valor da movimentação em termos de saldo
cbo2002ocupacao	Código da ocupação do trabalhador de acordo com a Classificação Brasileira de Ocupações (CBO 2002)
graudeinstrucao	Grau de instrução do trabalhador
idade	Idade do trabalhador
raçacor	Raça ou cor do trabalhador
Sexo	Sexo do trabalhador

```sql
CREATE TABLE MovCuritiba (
  competenciamov int DEFAULT NULL,
  secao varchar(10) DEFAULT NULL,
  subclasse int DEFAULT NULL,
  saldomovimentacao int DEFAULT NULL,
  cbo2002ocupacao int DEFAULT NULL,
  categoria int DEFAULT NULL,
  graudeinstrucao int DEFAULT NULL,
  idade int DEFAULT NULL,
  horascontratuais decimal(5,2) DEFAULT NULL,
  racacor int DEFAULT NULL,
  sexo int DEFAULT NULL,
  tipoempregador int DEFAULT NULL,
  tipoestabelecimento int DEFAULT NULL,
  tipomovimentacao int DEFAULT NULL,
  tipodedeficiencia int DEFAULT NULL,
  indtrabintermitente int DEFAULT NULL,
  indtrabparcial int DEFAULT NULL,
  salario decimal(10,2) DEFAULT NULL,
  tamestabjan int DEFAULT NULL,
  indicadoraprendiz int DEFAULT NULL,
  origemdainformacao int DEFAULT NULL,
  competenciadec int DEFAULT NULL,
  indicadordeforadoprazo int DEFAULT NULL,
  unidadesalariocodigo int DEFAULT NULL,
  valorsalariofixo decimal(10,2) DEFAULT NULL
)
```

Esta tabela é a mesma de Movimentacao, contudo, o nome é diferente e não contém as colunas uf e município pois contém apenas os registros de Curitiba.

```sql
CREATE TABLE autores (
    id_autor        INT AUTO_INCREMENT PRIMARY KEY,
    nome            VARCHAR(100) NOT NULL,
    nacionalidade   VARCHAR(50) DEFAULT 'Desconhecida',
    data_nascimento DATE
);
```

```sql
-- Tabela de livros
CREATE TABLE livros (
    id_livro        INT AUTO_INCREMENT PRIMARY KEY,
    titulo          VARCHAR(255) NOT NULL,
    ano_publicacao  INT,
    isbn            VARCHAR(13) UNIQUE, -- ISBN deve ser único
    id_autor        INT, -- Chave estrangeira para a tabela autores
    FOREIGN KEY (id_autor) REFERENCES autores(id_autor)
```

```
);
```

Desafio Final

Vamos para o desafio final para finalizar este curso?

http://200.17.199.250/siteprototipo/curso/mysql/qrmysql012.html