Apostila do Curso

Conteúdo e Atividades

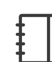
Banco de Dados com **MySQL**

Banco de dados com MySql

Nome:

Sobre o curso

O curso de MySQL foi desenvolvido para proporcionar ao aluno uma compreensão completa sobre o funcionamento, a estrutura e a manipulação de bancos de dados relacionais. Ao longo das aulas, o estudante aprenderá os conceitos fundamentais do Sistema de Gerenciamento de Banco de Dados (SGBD) MySQL, capacitando-o a interagir com dados de forma eficaz e profissional, desde a criação da estrutura do banco até a execução de consultas complexas.



Quantidade de Aulas
8 aulas

Carga horária
12 horas

Programas utilizados
Workbench

O que aprender com este curso?

O curso capacita o aluno a criar, gerenciar e consultar bancos de dados MySQL de forma profissional. O estudante aprenderá desde os fundamentos e o design de tabelas (DDL) até a manipulação completa de registros via operações CRUD. O conteúdo abrange técnicas avançadas de filtragem, ordenação, funções de agregação e agrupamentos para relatórios. Também são ensinados o uso de Junções (JOINs), subconsultas e operações de conjuntos como UNION. Por fim, o aluno dominará alterações estruturais em bancos existentes e as melhores práticas de gestão. Em suma, o curso habilita o planejamento e a configuração completa de ambientes de dados.

Sumário

1 - Introdução ao MySql
1.1 - O que é um Banco de Dados?
1.2 - A História dos Bancos de Dados
1.3 - O que um Banco de Dados pode resolver em uma Empresa?
1.4 - Por que Estudar Banco de Dados?
1.5 - O que é um SGBD?
1.6 - Tipos de SGBDs
1.7 - O que é SQL?
1.8 - A História do SQL
1.9 - O que é MySQL?
1.10 - A História e Popularidade do MySQL
1.11 - O que é MySQL Workbench?
1.12 - Por que Usar MySQL Workbench?
1.13 - Passo a Passo da Instalação
1.14 - O que é o Usuário 'Root'?
1.15 - Por que Usar o 'Root' para Conectar?
1.16 - Conectando ao Servidor Local MySQL
1.17 - Entendendo SCHEMAS e Query
1.18 - CREATE SCHEMA vs. CREATE DATABASE
1.19 - Executando o Comando de Criação
1.20 - Visualizando o Banco de Dados Criado
1.21 - Exercício prático
2 - Tipos de Dados e Criação de Tabelas
2.1 - Introdução às Tabelas no MySQL
2.2 - Estrutura de uma Tabela: Colunas e Linhas
2.3 - Por que Usamos Tabelas?
2.4 - Tipos de Dados no MySQL
2.5 - O que é um Registro?
2.6 - Criando sua Primeira Tabela no MySQL Workbench
2.7 - Entendendo os Comandos SQL: Convenções de Escrita
2.8 - Detalhando a Criação da Tabela 'alunos'
2.8.1 - Coluna id : INT AUTO_INCREMENT PRIMARY KEY
2.8.2 - Coluna nome : VARCHAR( ) NOT NULL
2.8.3 - Coluna nota : NUMERIC(5,2)
2.8.4 - Coluna ativo : BOOLEAN DEFAULT TRUE
2.8.5 - Coluna criado_em : TIMESTAMP DEFAULT CURRENT_TIMESTAMP
2.9 - Executando o Comando de Criação da Tabela
2.9.1 - Executando o Comando:
2.10 - Exercício Prático
2.10.1 - Instruções:
3 - CRUD Básico
3.1 - O que é CRUD?
3.1.1 - A Importância do CRUD
3.2 - Os Comandos do CRUD na Prática
3.2.1 - C - Create (Criar)
3.2.2 - Comando INSERT INTO
3.2.2.0.1 - Sintaxe Básica:
3.2.3 - Exemplo Prático e Explicação Detalhada
3.2.3.0.1 - Comando:
3.2.4 - Explicação:
3.2.5 - Um pouco mais a fundo:
3.2.6 - História para ilustrar:
3.2.7 - R - Read (Ler)
3.2.8 - Comando SELECT
3.2.9 - Sintaxe Básica:
3.2.10 - Exemplo Prático e Explicação Detalhada
3.2.11 - História para ilustrar:
3.2.12 - U - Update (Atualizar)
3.2.13 - Comando UPDATE
3.2.14 - Exemplo Prático e Explicação Detalhada
3.2.15 - A Importância do WHERE:
3.2.16 - D - Delete (Excluir)
3.2.17 - Comando DELETE
3.2.18 - MySQL Workbench: Preparando o Ambiente
3.2.19 - Prática Guiada: CRUD Completo
3.2.20 - Criando um Novo Registro (CREATE)
3.2.21 - Consultando o Registro Criado (READ)
3.2.22 - Atualizando o Registro (UPDATE)
3.2.23 - Confirmando a Atualização (READ)
3.2.24 - Confirmando a Exclusão (READ)
3.3 - Exercícios práticos
3.4 - Conclusão
4 - Filtragem, Ordenação e limitação
4.1 - Filtragem de Dados com WHERE
4.1.1 - Comparadores Simples
4.1.2 - Intervalo de Valores com BETWEEN...AND
4.1.3 - Lista de Valores com IN e NOT IN
4.1.4 - Padrões de Texto com LIKE
4.1.5 - Combinação Lógica com AND, OR e NOT
4.1.6 - Valores Ausentes com IS NULL e IS NOT NULL
4.2 - Ordenação de Dados com ORDER BY
4.2.1 - Ordenação Básica: ASC e DESC

4.2.2 - Tratamento de Valores Nulos na Ordenação
4.2.3 - Ordenação Sensível ou Insensível a Maiúsculas/Minúsculas
4.3 - Limitação de Dados com LIMIT
4.4 - A Importância da Combinação de Filtragem, Ordenação e Limitação
4.5 - Atividade prática
5 - Agregação e Agrupamento
5.1 - Agregação e Agrupamento: Conceitos Fundamentais
5.1.1 - O que são Agregação e Agrupamento?
5.1.2 - Funções Agregadas Básicas
5.1.3 - Agrupamento com GROUP BY
5.1.4 - Filtragem: WHERE vs. HAVING
5.1.5 - Comportamento com Valores NULL e COUNT
5.1.6 - Agrupamento por Múltiplas Colunas
5.1.7 - Função FLOOR()
5.2 - Prática no MySQL Workbench
5.2.1 - Configuração Inicial
5.2.2 - Funções Agregadas Básicas na Prática
5.2.3 - Agrupamento na Prática
5.2.4 - WHERE vs. HAVING na Prática
5.2.5 - NULL e COUNT na Prática
5.2.6 - Agrupamento por Múltiplas Colunas e FLOOR na Prática
5.3 - Atividade prática da apostila
6 - Joins
6.1 - O que são JOINS?
6.1.1 - Cenário de Exemplo: Alunos e Matrículas
6.2 - A Importância da Cláusula ON nos JOINS
6.2.1 - Exemplo Prático da Cláusula ON
6.2.2 - Por que a Cláusula ON é tão importante?
6.3 - Tipos de JOINS no MySQL
6.3.1 - INNER JOIN (ou apenas JOIN)
6.3.2 - LEFT JOIN (ou LEFT OUTER JOIN)
6.3.3 - RIGHT JOIN (ou RIGHT OUTER JOIN)
6.3.4 - CROSS JOIN (Produto Cartesiano)
6.3.5 - SELF JOIN (Auto-Join)
6.3.6 - FULL OUTER JOIN (Simulação no MySQL)
6.4 - Desafio Prático: Criando e Manipulando Tabelas com JOINS
6.4.1 - Preparando o Ambiente: MySQL Workbench
6.4.2 - Criando a Tabela matriculas
6.4.3 - Adicionando Registros à Tabela matriculas
6.4.4 - Aplicando os Comandos JOINS na Prática
6.4.4.1 - INNER JOIN
6.4.4.2 - LEFT JOIN
6.4.4.3 - RIGHT JOIN
6.4.4.4 - CROSS JOIN
6.4.4.5 - SELF JOIN
6.5 - Atividades Práticas no MySQL Workbench
6.5.1 - Criação do Banco de Dados
6.5.2 - Criação das Tabelas
6.5.3 - Inserção de Dados
6.5.4 - Atividades Práticas com JOINs
6.5.4.1 - Atividade: INNER JOIN
6.5.4.2 - Atividade: LEFT JOIN
6.5.4.3 - Atividade: RIGHT JOIN
6.5.4.4 - Atividade: CROSS JOIN
6.5.4.5 - Atividade: SELF JOIN
7 - Subconsultas e Conjuntos
7.1 - Subconsultas (Subqueries) no MySQL
7.1.1 - O que são Subconsultas?
7.1.2 - Tipos Comuns de Subconsultas
7.1.2.1 - Subconsulta no WHERE com IN
7.1.2.2 - Subconsulta com EXISTS
7.1.2.3 - Subconsulta Escalar
7.1.2.4 - Subconsulta no FROM (Tabela Derivada)
7.1.3 - Subconsultas Correlacionadas vs. Não Correlacionadas
7.1.4 - Vantagens e Cuidados com Subconsultas
7.2 - Operadores de Conjunto no MySQL
7.2.1 - UNION e UNION ALL
7.2.2 - Regras Importantes para UNION/UNION ALL
7.2.3 - Outros Operadores de Conjunto (INTERSECT e EXCEPT/MINUS)
7.3 - Prática no MySQL Workbench
7.3.1 - Abrindo o MySQL Workbench
7.3.2 - Comandos de Subconsultas
7.3.2.1 - Subconsulta no WHERE com IN
7.3.2.2 - Subconsulta com EXISTS
7.3.2.3 - Subconsulta Escalar no SELECT
7.3.2.4 - Subconsulta no FROM (Tabela Derivada)
7.3.2.5 - Subconsulta Correlacionada
7.3.3 - Comandos de Operadores de Conjunto
7.3.3.1 - UNION (Remove Duplicatas)
7.3.3.2 - UNION ALL (Mantém Duplicatas)
7.4 - Exercício prático de Subconsultas e conjuntos
7.4.1 - Preparação do Ambiente
7.4.1.1 - Crie e selecione o banco de dados:
7.4.1.2 - Crie a tabela funcionarios:
7.4.1.3 - Crie a tabela projetos_alocados:
7.4.2 - Inserindo os Dados
7.4.2.1 - Insira os dados na tabela funcionarios:
7.4.2.2 - Insira os dados na tabela projetos_alocados:
7.4.3 - Hora do Desafio - Realizando as Consultas
7.4.3.1 - Desafio 1: Subconsulta no WHERE com IN
7.4.3.2 - Desafio 2: Subconsulta Escalar no SELECT

7.4.3.3 - Desafio 3: Subconsulta no FROM (Tabela Derivada)
7.4.3.4 - Desafio 4: Conjuntos com UNION
8 - Alterações DDL
8.1 - O que são Alterações DDL?
8.1.1 - Data Definition Language (DDL)
8.1.2 - Por que usar ALTER e fazer mudanças DDL?
8.2 - Principais Tipos de Alteração DDL
8.2.1 - Adicionar uma Coluna
8.2.2 - Remover uma Coluna
8.2.3 - Modificar o Tipo ou Definição de uma Coluna
8.2.4 - Entendendo TRUNCATE: A História do Caderno
8.3 - Adicionar / Remover Chave Primária
8.3.1 - Adicionar / Remover Índice (Index)
8.3.2 - Adicionar / Remover Chave Estrangeira (Foreign Key)
8.3.3 - Renomear Tabela
8.3.4 - Trocar Engine, Charset ou Collation
8.3.5 - Definir/Alterar DEFAULT, NOT NULL, UNIQUE
8.4 - Efeitos Colaterais e Cuidados ao Utilizar DDL
8.5 - ROLLBACK e Transações no MySQL
8.5.1 - A História do Checkpoint no Videogame
8.5.2 - Transações (TRANSACTION)
8.5.3 - Comandos de Transação
8.6 - Prática no MySQL Workbench
8.6.1 - Adicionar uma Coluna
8.6.2 - Remover uma Coluna
8.6.3 - Modificar ou Renomear uma Coluna
8.6.4 - TRUNCATE (Apagar Todos os Dados da Tabela)
8.6.5 - Adicionar / Remover Chave Primária
8.6.6 - Adicionar / Remover Índice (Index)
8.6.7 - Adicionar Chave Estrangeira (Foreign Key)
8.6.8 - Renomear Tabela
8.6.9 - Trocar Engine, Charset e Collation
8.6.10 - Definir/Alterar DEFAULT, NOT NULL, UNIQUE
8.7 - Atividade Prática

1. Introdução ao MySql

Seja muito bem-vindo(a), futuro(a) especialista em Banco de Dados! É uma grande satisfação tê-lo(a) conosco nesta jornada de aprendizado sobre MySQL. Se você chegou até aqui, é porque já compreende a importância de dominar os bancos de dados no cenário tecnológico atual. Prepare-se para desvendar os segredos do armazenamento, organização e recuperação de informações de forma eficiente, e para construir uma base sólida que impulsionará sua carreira. Vamos começar nossa aula entendendo os conceitos fundamentais que regem o mundo dos dados.

1.1. O que é um Banco de Dados?

Um banco de dados é um sistema que permite armazenar, organizar e recuperar informações de modo eficiente. Os dados são guardados em estruturas chamadas tabelas, compostas por linhas (registros) e colunas (campos). Cada linha representa uma entidade (como um cliente ou produto), e as colunas representam características dessa entidade (como nome, idade, preço etc.). O sistema responsável por gerenciar esses dados é chamado de Sistema Gerenciador de Banco de Dados (SGBD), que controla inserções, consultas, segurança, integridade e desempenho.

1.2. A História dos Bancos de Dados

Desde os primeiros tempos, a humanidade registrou informações em papéis, pergaminhos e fichários. Com o surgimento dos computadores na década de 1960, começaram os bancos de dados informatizados, mas ainda baseados em modelos rígidos como o hierárquico ou em rede, que exigiam navegação por ponteiros e pouco flexíveis.

Em 1970, Edgar Frank Codd, cientista da IBM, publicou o artigo seminal “A Relational Model of Data for Large Shared Data Banks”, propondo o modelo relacional, que trata os dados como tabelas lógicas, indiferentes à forma física de armazenamento. A partir desse modelo vieram projetos como o System R da IBM (1974–1977), que implementou a linguagem SQL (antes chamada SEQUEL) e tornou viável o modelo relacional na prática. Outro projeto pioneiro foi o Ingres, desenvolvido na Universidade da Califórnia, que deu origem a diversos bancos comerciais como Sybase e SQL Server.

Já em 1976, Peter Chen propôs o modelo Entidade-Relacionamento (ER), que trouxe uma forma clara de planejar a estrutura de dados com entidades e relacionamentos. Na década de 1980, bancos como Oracle, IBM DB e Microsoft SQL Server se consolidaram comercialmente, popularizando o uso em empresas e setores diversos.

1.3. O que um Banco de Dados pode resolver em uma Empresa?

Bancos de dados são ferramentas poderosas para resolver diversos desafios empresariais, proporcionando:

- **Organização e Estrutura:** Com tabelas bem definidas, um banco de dados evita

redundância e inconsistência nos dados. A aplicação de normalização (divisão em tabelas menores) garante que a informação seja armazenada de forma eficiente e consistente.

- **Consultas e Manipulações Eficientes:** A linguagem SQL permite ao usuário expressar o que deseja (por exemplo: listar todos os empréstimos atrasados), e o SGBD decide como buscar os dados de forma otimizada. Consultas SQL podem combinar tabelas, filtrar, agrupar e ordenar os dados de forma muito eficiente.

- **Integridade e Segurança dos Dados:** Os bancos de dados suportam restrições como unicidade, integridade referencial, campos obrigatórios e regras de negócios diretamente no esquema. Isso impede duplicação, dados inválidos ou inconsistentes. Além disso, controlam quem pode acessar ou modificar cada dado.

- **Transações Confiáveis:** Operações que envolvem múltiplas etapas (como transferir dinheiro de uma conta para outra) são tratadas como transações, que obedecem às propriedades ACID (Atomicidade, Consistência, Isolamento, Durabilidade). Garantindo que tudo aconteça corretamente ou, em caso de falha, volte ao estado anterior sem bagunça nos dados.

- **Concorrência e Desempenho:** Em ambientes com muitos usuários acessando dados ao mesmo tempo, técnicas como MVCC (Controle de Concorrência Multiversão) permitem que cada usuário veja uma cópia consistente dos dados, sem bloqueios que prejudicam a performance. Além disso, o uso de índices acelera buscas, evitando digitalizações de toda a tabela.

1.4. Por que Estudar Banco de Dados?

Desde sistemas financeiros e redes sociais até aplicações escolares, bancos de dados robustos são essenciais para sistemas confiáveis, seguros e eficientes. Compreender sua evolução histórica, o modelo relacional, SQL, integridade, transações e otimização de consultas fornece uma base sólida para projetar sistemas reais de maneira profissional. Isso transforma dados em informação útil e possibilita decisões bem fundamentadas.

1.5. O que é um SGBD?

Um SGBD, ou Sistema Gerenciador de Banco de Dados, é um conjunto de programas de computador que administra um ou vários bancos de dados. Ele age como intermediário entre as aplicações (como sistemas de cadastro, relatórios ou ERPs) e os dados armazenados, liberando o desenvolvedor da tarefa de controlar diretamente acesso, persistência e manipulação dos dados.

O SGBD gerencia operações fundamentais como criar, consultar, alterar e excluir dados em bancos de dados, garantindo que esses processos ocorram de forma organizada e segura. Ele também fornece uma interface, geralmente por meio de APIs ou drivers, que executam comandos SQL ou outros tipos de linguagens de consulta para que as aplicações possam incluir, alterar ou consultar dados sem lidar com o armazenamento físico.

1.6. Tipos de SGBDs

Existem diferentes modelos de SGBDs, cada um adequado a necessidades específicas. Os principais tipos são:

- **Relacional (RDBMS):** Organiza os dados em tabelas relacionadas entre si por chaves, sendo o tipo mais utilizado hoje. Suporta SQL e exige um esquema predefinido.

- **Não-relacional (NoSQL):** Armazena dados em formatos como documentos, chave-valor, colunas ou grafos. É flexível em esquemas e indicado para grandes volumes de dados distribuídos.

- **Orientado a objetos (SGBDO/O):** Trata os dados como objetos, mantendo identidade, métodos e heranças, ideal para aplicações com entidades complexas.

- **Objeto-relacional (SGBDOR):** Combina tabelas relacionais com recursos de orientação a objetos, permitindo tipos de dados personalizados e herança dentro do modelo relacional.

Além desses, há ainda SGBDs com modelos hierárquicos, em rede, multidimensional e em memória, conforme as necessidades de desempenho ou estrutura dos dados.

1.7. O que é SQL?

SQL (Structured Query Language ou Linguagem de Consulta Estruturada) é a linguagem padrão usada para criar, consultar, modificar e gerenciar dados em bancos de dados relacionais.

1.8. A História do SQL

A SQL surgiu no início da década de 1970 nos laboratórios da IBM, com Donald D. Chamberlin e Raymond F. Boyce. Inicialmente chamada de SEQUEL ("Structured English Query Language"), a linguagem foi renomeada para SQL porque SEQUEL já era marca registrada. O objetivo era proporcionar uma sintaxe que permitisse expressar consultas de forma clara e em inglês estruturado, manipulando dados conforme o modelo relacional de Edgar F. Codd, implementado no projeto System R da IBM. Hoje, SQL é a linguagem universal em bancos de dados relacionais, ainda que cada sistema adote variações que estendem ou adaptam o padrão.

1.9. O que é MySQL?

MySQL é um Sistema Gerenciador de Banco de Dados Relacional (RDBMS) que implementa a linguagem SQL para armazenar, buscar e manipular dados.

1.10. A História e Popularidade do MySQL

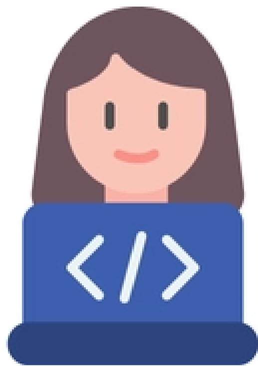
O MySQL foi criado em 1994 pela empresa sueca MySQL AB, fundada por David Axmark, Allan Larsson e Michael “Monty” Widenius. A sua primeira versão pública foi lançada em maio de 1995. O nome MySQL combina “My”, que é o nome da filha de Widenius, com “SQL” a linguagem que o sistema utiliza.

Desde o início, o MySQL foi distribuído sob a licença Open Source (GPL), permitindo seu uso gratuito, além de ser instalado em Windows, Linux, macOS e outros sistemas operacionais. Em janeiro de 2008, a MySQL AB foi adquirida pela Sun Microsystems, que por sua vez foi comprada pela Oracle Corporation em 2009. Desde então, o MySQL continua disponível como código aberto, embora existam versões comerciais com suporte adicional.

A origem do MySQL está ligada à insatisfação com o desempenho de sistemas anteriores como o mSQL e os arquivos ISAM, considerados lentos ou pouco flexíveis. Michael Widenius e seus colegas decidiram criar um sistema compatível com a API domSQL, mas com interface SQL e melhor desempenho. Com isso, permitiram que desenvolvedores migrassem facilmente para MySQL sem grandes adaptações no código.

O MySQL ganhou popularidade por sua facilidade de uso, velocidade e integração com tecnologias web, especialmente com PHP e o pacote LAMP (Linux, Apache, MySQL, PHP/Python/Perl). Isso fez do MySQL a escolha natural para muitos sites dinâmicos e sistemas de gestão de conteúdo como WordPress, Drupal, Joomla, além de grandes empresas como Facebook, Twitter e Wikipedia que utilizam MySQL em produção.

1.11. O que é MySQL Workbench?



O MySQL Workbench é uma ferramenta gráfica oficial da Oracle que combina em um único ambiente a modelagem de dados, desenvolvimento em SQL e administração de servidores MySQL.

1.12. Por que Usar MySQL Workbench?

Ele funciona como um “painel central” onde você pode:

- **Modelar visualmente seu banco de dados:** com diagramas (ERDs) criando e ajustando tabelas, chaves estrangeiras, índices, etc., tanto a partir do zero quanto a partir de um banco já existente.
- **Escrever, editar e executar comandos SQL:** com ajuda de destaque de sintaxe, auto-completar e histórico, facilitando aprendizado e produtividade.
- **Administrar o servidor e banco de dados:** acessando usuários, permissões, backups, logs e monitoramento de desempenho, tudo via interface.

O MySQL Workbench é o ambiente perfeito para aprender banco de dados MySQL de forma completa e prática. Ele facilita a visualização da estrutura, o domínio do SQL e a gestão do servidor, tornando nossa jornada de estudos muito mais eficaz e envolvente.

1.13. Passo a Passo da Instalação

Para instalar o MySQL Workbench no Windows, siga os passos abaixo:

1. **Abrir o Navegador:** Comece abrindo seu navegador de internet.
2. **Pesquisar o Site:** Na barra de pesquisa, digite "www.mysql.com/downloads" pressione Enter.
3. **Localizar o Download:** Role a página para baixo até encontrar a seção de downloads e clique em "MySQL Community (GPL) Downloads", destacado em azul.
4. **Selecionar o Instalador:** Na próxima página, clique em "MySQL Installer for Windows".
5. **Baixar a Versão Mais Recente:** Clique em "Download" na última versão disponível (ex: 8.0.43) no primeiro da lista.
6. **Ignorar Login:** Quando solicitado a fazer login, clique em "No thanks, just start my download." para iniciar o download diretamente.
7. **Executar o Instalador:** Após o download, clique duas vezes no arquivo .exe baixado para iniciar o instalador.
8. **Escolher Tipo de Instalação:** No instalador do MySQL, selecione a opção "Full" e clique em "Next".
9. **Confirmar Produtos:** Será exibida a lista de todos os produtos MySQL a serem instalados. Clique em "Execute".
10. **Avançar:** Após o download de todos os produtos, clique em "Next".
11. **Instalação dos Produtos:** O instalador realizará a instalação de cada produto. Clique em "Next" novamente.
12. **Configuração Inicial:** Clique em "Next" nas

próximas telas de configuração, sem alterar nada.
13. **Definir Senha do Root:** Neste local, será solicitada a senha do usuário "Root". Defina a senha como "123456" nos dois campos e clique em "Next".
14. **Serviço MySQL:** A tela seguinte informa sobre o serviço do MySQL no Windows. Não altere nada e clique em "Next".
15. **Configurações Finais:** Clique em "Next" nas próximas telas de configuração.
16. **Aplicar Configurações:** Clique em "Execute" para aplicar as configurações.
17. **Finalizar:** Após a aplicação das configurações, clique em "Finish".
18. **Configuração do Router**: Será exibida uma configuração de Router que não será utilizada. Clique em "Finish".
19. **Testar Conexão:** Será necessário testar a conexão com o MySQL. Coloque a senha "123456" e clique em "Check" para verificar a conexão. Se estiver tudo certo, clique em "Next".
20. **Executar Configuração:** Confirme as configurações clicando em "Execute".
21. **Finalizar Configuração:** Após a conclusão, clique em "Finish".
22. **Concluir Instalação:** Clique em "Next" e, por fim, em "Finish" para encerrar a instalação do MySQL.

Após a instalação, o MySQL Workbench será aberto automaticamente.

1.14. O que é o Usuário 'Root'?

Em MySQL, o usuário root é a conta padrão criada durante a instalação. Ele é o super-usuário do banco de dados, com "poder divino". Tem acesso completo a todos os bancos, tabelas, operações e privilégios, sem restrições. Diferente do usuário root do sistema operacional, este root existe apenas dentro do próprio MySQL.

1.15. Por que Usar o 'Root' para Conectar?

O usuário root nos oferece total autonomia para criar, gerenciar e explorar o banco de dados sem barreiras, sendo ideal para fins de estudo e desenvolvimento inicial.

1.16. Conectando ao Servidor Local MySQL

Com o MySQL Workbench aberto, clique em "local mysql" onde consta nosso servidor local MySQL que testamos durante a instalação. Antes de conectar, será solicitada a senha para acessarmos nosso MySQL local. Clique em "Password", coloque a senha que definimos durante o curso (123456) e clique em "OK" para conectar.

1.17. Entendendo SCHEMAS e Query

Ao conectar, você verá o painel "SCHEMAS" e "Query ".

- **Schema:** É sinônimo de banco de dados. São coleções de objetos (tabelas, views, rotinas, etc.) organizados em um mesmo namespace. Quando você cria um schema (banco de dados), está basicamente criando um contêiner para guardar suas tabelas, definir relacionamentos e aplicar restrições.

- **Query:** Uma Query é uma consulta SQL, um comando que você escreve para solicitar ou manipular dados no banco. O painel "Query " será o local onde você executará seus comandos SQL no MySQL.

1.18. CREATE SCHEMA vs. CREATE DATABASE

Quando criamos um banco de dados, podemos utilizar dois métodos:

- **CREATE DATABASE:** É mais comum em bancos como MySQL, SQL Server e SQLite.

- **CREATE SCHEMA**: Tem origem no padrão SQL ANSI, e é mais utilizado em bancos como PostgreSQL e Oracle, onde schema é uma estrutura dentro de um banco de dados, e não o banco em si.

Em MySQL, CREATE SCHEMA e CREATE DATABASE são sinônimos e podem ser usados de forma intercambiável.

1.19. Executando o Comando de Criação

Para criar seu primeiro banco de dados, no campo da "Query ", escreva o comando:

```sql
CREATE SCHEMA curso;
```

Para executar o comando, clique no primeiro símbolo de raio no menu superior da Query. Após a execução, você verá a confirmação do banco de dados "curso" criado.

1.20. Visualizando o Banco de Dados Criado

Após criar o banco de dados, para que ele apareça no painel "SCHEMAS", clique no símbolo das setas girando entre si (ícone de atualização) ao lado do painel "SCHEMAS". Agora sim, seu banco de dados curso estará visível.

1.21. Exercício prático

Com o MySQL Workbench aberto, crie um um novo banco de dados utilizando **DATABASE** ou **SCHEMA** com o nome (exerc_apostila_um) dentro da query 1.

Navigator
SCHEMAS
Filter objects
exerc_apostila_um

Seu banco de dados criado deverá constar como a imagem acima.

Anotações

Seja bem-vindo(a) à nossa segunda apostila do curso de Banco de Dados com MySQL sobre Tipos de dados e criação de Tabelas.

2.1. Introdução às Tabelas no MySQL

Imagine que você tem um caderno de anotações onde registra informações sobre seus amigos: nome, idade, telefone e endereço. Cada linha do caderno representa um amigo, e cada coluna representa uma informação específica sobre ele.

No MySQL, uma **tabela** funciona de maneira muito semelhante. Ela é a estrutura fundamental onde armazenamos dados organizados em linhas e colunas. Pense nela como uma planilha gigante, mas muito mais poderosa e eficiente para gerenciar grandes volumes de informações.

- **Cada linha** da tabela representa um **registro** único (como um amigo no seu caderno).
- **Cada coluna** da tabela representa um **tipo de informação** específica (como nome, idade, etc.).

As tabelas são a espinha dorsal de qualquer banco de dados relacional, permitindo que os dados sejam armazenados de forma lógica e acessível. Compreender o funcionamento das tabelas é o primeiro passo para dominar o MySQL e construir sistemas robustos.

2.2. Estrutura de uma Tabela: Colunas e Linhas

Para entender melhor como uma tabela funciona no MySQL, vamos detalhar seus dois componentes principais:

1. **Colunas (Campos):** Pense nas colunas como as categorias de informação que você deseja armazenar. Cada coluna é projetada para guardar um tipo específico de dado. Por exemplo, em uma tabela de "clientes", você pode ter colunas como:

- **id_cliente:** Um número único para identificar cada cliente.
- **nome:** O nome completo do cliente.
- **e-mail:** O endereço de e-mail do cliente.
- **telefone:** O número de telefone do cliente.

1. **Linhas (Registros)**: As linhas, por outro lado, são as entradas individuais de dados. Cada linha contém um conjunto completo de informações para um único item ou entidade. Usando o exemplo da tabela de "clientes":

- **Linha 1:** id_cliente: 1, nome: João Silva, email: joão@email.com, telefone: (51) 99999-9999
- **Linha 2:** id_cliente: 2, nome: Maria Oliveira, email:maria@email.com, telefone: (51) 988888-8888

Cada linha é um "registro" completo, e todas as linhas em uma tabela seguem a mesma estrutura de colunas. Essa organização padronizada é o que torna os bancos de dados relacionais tão eficientes para gerenciar informações.

2.3. Por que Usamos Tabelas?

As tabelas são a base de qualquer banco de dados relacional e são essenciais por diversas razões. Elas permitem organizar grandes quantidades de informações de forma estruturada e eficiente, o que é crucial para qualquer sistema que lide com dados. Veja os principais motivos:

- **Armazenar Dados de Maneira Organizada:** As tabelas garantem que os dados sejam guardados de forma padronizada, facilitando o acesso, a busca e a atualização das informações. Imagine tentar encontrar um dado específico em um arquivo sem nenhuma organização, seria um caos! As tabelas resolvem isso.

- **Relacionar Informações:** Uma das maiores vantagens dos bancos de dados relacionais é a capacidade de conectar diferentes tabelas. Por exemplo, em uma loja online, você pode ter uma tabela de clientes e outra de pedidos. As tabelas permitem associar um cliente específico a todos os seus pedidos, criando uma rede de informações interligadas.

- **Realizar Consultas Rápidas:** Com os dados bem organizados em tabelas, é possível realizar consultas complexas e encontrar informações específicas de forma muito rápida. Quer saber todos os clientes que moram em uma determinada cidade? Uma consulta bem feita pode trazer essa informação em segundos.

Entender o que são tabelas no MySQL é fundamental para trabalhar com bancos de dados. Elas são a base onde armazenamos e organizamos informações, permitindo que sistemas e aplicativos funcionem de maneira eficiente e escalável.

2.4. Tipos de Dados no MySQL

Quando você começa a criar suas tabelas no MySQL, cada coluna exige que você defina um **tipo de dado**. Essa definição é crucial, pois ela comunica ao banco de dados duas informações muito importantes:

- **Que tipo de informação será armazenada:** Se é um número, um texto, uma data, um valor verdadeiro/falso, etc.

- **Como o MySQL deve lidar com ela:** Isso inclui como o dado será armazenado, o espaço que ocupará, e como ele pode ser manipulado em operações e consultas.

Escolher o tipo de dado correto ajuda o banco a proteger a integridade dos dados, acelerar pesquisas e evitar desperdício de espaço. Imagine uma tabela onde você quer guardar dados sobre alunos:

- Se a coluna é para o **número de matrícula**, você usará um tipo numérico apropriado (ex: INT).

- Se é para o **nome do aluno**, deve usar um tipo de texto (ex: VARCHAR).

- Se é para a **data de aniversário**, existe um tipo específico para datas (ex: DATE).

Cada tipo de dado tem suas próprias regras e limites. Por exemplo:

Número:

- **INT** : Armazena apenas números inteiros (sem casas decimais).
- **DECIMAL** (ou **NUMERIC**): Armazena números com casas decimais exatas, útil para valores como preços ou notas, onde a precisão é fundamental.

Textos:

- **VARCHAR**: Serve para textos curtos e que mudam de tamanho, como nomes ou e-mails. O VAR significa “variável”, ou seja, o banco não reserva espaço fixo para cada registro, economizando espaço.

- **TEXT:** Usado quando se espera guardar textos longos, por exemplo, descrições completas ou artigos.

Valores Pré-definidos:

- **ENUM:** Se você sabe que sempre haverá apenas algumas opções de valor (ex: status 'ativo', 'inativo'), pode usar ENUM , que limita as entradas a uma lista definida.

Datas e Horários:

- **DATE:** Para guardar apenas a data no formato YYYY-MM-DD.
- **DATETIME:** Para guardar data e hora juntas.
- **TIMESTAMP:** Similar ao **DATETIME**, mas trata fusos horários automaticamente e é ideal para registrar o momento exato de inserções ou alterações no banco.

Escolher o tipo de dado certo faz uma grande diferença: evita erros na hora de salvar ou calcular valores, torna as buscas por informações mais rápidas e permite que o banco use menos memória. Não se preocupe em memorizar todos agora; a prática o ajudará a entender qual usar em cada situação.

2.5. O que é um Registro?

Em uma tabela do MySQL, um **registro** (também chamado de **linha** ou row, e às vezes de tupla) representa uma única entrada. Pense nele como um conjunto de informações relacionadas que pertence a uma pessoa, objeto ou evento específico.

Cada registro agrupa valores em colunas distintas. Por exemplo, em uma tabela de alunos , cada registro pode conter:

- **ID:** O identificador único do aluno.
- **nome:** O nome completo do aluno.
- **idade**: A idade do aluno.
- **turma:** A turma em que o aluno está matriculado.
- **data_de_matricula:** A data em que o aluno foi matriculado.

Quando você insere dados em uma tabela, você está adicionando um novo registro. Ao consultar, você está buscando um ou mais registros. Cada coluna desse registro espera um valor compatível com o tipo de dado definido para ela, e juntos, esses valores formam uma linha lógica na tabela.

A estrutura de cada registro é igual em toda a tabela, o que significa que todos os registros têm as mesmas colunas, embora com valores diferentes. Para entender de forma simples, imagine uma prateleira de fichas de papel, onde cada ficha possui campos preenchidos como nome, data de nascimento, telefone, etc. No contexto do MySQL, cada ficha equivale a um registro armazenado em uma linha da tabela. E assim como cada ficha tem o mesmo formato, cada registro na tabela segue a mesma estrutura definida pelas colunas.

2.6. Criando sua Primeira Tabela no MySQL Workbench

Agora que já revisamos os conceitos fundamentais, vamos colocar a mão na massa e criar nossa primeira tabela no MySQL Workbench. Esta ferramenta é o seu principal aliado para interagir com o banco de dados de forma visual e prática.

Passo a Passo:

1. **Abrindo o MySQL Workbench:** Você pode abrir o MySQL Workbench pesquisando por ele na barra de pesquisa do Windows ou através do ícone na sua área de trabalho.
2. **Conectando ao Serviço Local:** Com o Workbench aberto, você verá uma lista de conexões. Clique na sua conexão local, geralmente identificada como Local instance MySQL ou similar, dentro de MySQL Connections.
3. **Inserindo a Senha:** Ao tentar acessar seu

serviço MySQL local, será solicitada a senha que você definiu durante a instalação. Digite a senha (no nosso exemplo, 123456) e clique em "OK".
4. **Dentro do Serviço Local:** Parabéns! Você está agora dentro do serviço local do MySQL Workbench, pronto para começar a trabalhar com seus bancos de dados e tabelas.

2.7. Entendendo os Comandos SQL: Convenções de Escrita

Antes de mergulharmos na criação da tabela, é importante entender uma convenção comum ao escrever comandos SQL. Você deve ter notado que, ao longo do curso, utilizamos palavras-chave como CREATE TABLE , INSERT INTO e SELECT em letras maiúsculas.

Embora o SQL não exija que essas palavras estejam em maiúsculas (o SQL não é case-sensitive para comandos), essa prática é amplamente adotada por facilitar a leitura e organização do código, especialmente para quem está iniciando ou revisitando scripts depois de um tempo. Manter essa convenção torna o código mais legível e padronizado, o que é uma boa prática no desenvolvimento de bancos de dados.

2.8. Detalhando a Criação da Tabela 'alunos'

Vamos agora criar a tabela **alunos** e entender cada parte do comando SQL. Esta tabela armazenará informações básicas sobre os alunos de uma instituição. No MySQL Workbench, com seu Schema curso selecionado (clique com o botão direito em curso no painel **SCHEMAS** e selecione **Set as Default Schema**), digite o seguinte comando na área da **Query 1**:

```sql
CREATE TABLE alunos (
id INT AUTO_INCREMENT PRIMARY KEY,
nome VARCHAR(100) NOT NULL,
nota NUMERIC(5,2),
ativo BOOLEAN DEFAULT TRUE,
criado_em TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
);
```

Agora, vamos detalhar cada linha deste comando:

2.8.1. Coluna id : INT AUTO_INCREMENT PRIMARY KEY

- **id:** É o nome da nossa coluna, que servirá como identificador único para cada aluno. Por exemplo, o ID 21 pertence ao aluno João. Isso facilita a identificação e futuras consultas.
- **INT:** Define o tipo de dado da coluna como integer , ou seja, armazenará números inteiros (sem casas decimais). É perfeito para identificadores, pois são eficientes e fáceis de usar.
- **AUTO_INCREMENT:** Esta propriedade diz ao MySQL para gerar automaticamente um número único para esta coluna sempre que um novo registro for inserido. Por padrão, ele começa com 1 e aumenta em 1 a cada novo aluno. Isso significa que você não precisa se preocupar em digitar manualmente o próximo ID.
- **PRIMARY KEY:** Define esta coluna como a **chave primária** da tabela. Uma chave primária é um identificador exclusivo para cada registro. Isso garante que nenhum outro registro possa ter o mesmo valor no campo id , assegurando que cada aluno seja único e facilmente localizável. A combinação AUTO_INCREMENT com PRIMARY KEY é muito comum e prática para criar identificadores únicos.

2.8.2. Coluna nome : VARCHAR( ) NOT NULL

- **nome:** Nome da coluna, que armazenará o nome completo do aluno.
- **VARCHAR(100):** Define o tipo de dado como VARCHAR , que é usado para armazenar palavras ou textos curtos. O número 100 entre parênteses indica o limite máximo de caracteres que podem ser armazenados (neste caso, até 100 caracteres). O VAR em VARCHAR significa "variável", o que é muito eficiente, pois o banco de dados só usa o espaço necessário para o texto inserido, economizando memória.
- **NOT NULL:** Esta é uma restrição que impede que a coluna fique vazia. Ou seja, sempre que você for cadastrar um novo aluno, é obrigatório informar um valor para o nome . Se tentar inserir um registro sem preencher esta coluna, o MySQL impedirá a operação e mostrará um erro, garantindo a integridade dos dados.

2.8.3. Coluna nota : NUMERIC(5,2)

- **nota:** Nome da coluna, para armazenar a nota do aluno.
- **NUMERIC(5,2):** Define o tipo de dado como NUMERIC , que é ideal para números com precisão fixa, como notas ou valores monetários. 5 (precisão) indica o número total de dígitos permitidos (incluindo os antes e depois da vírgula), e o 2 (escala) indica o número de casas decimais após a vírgula. Assim, valores como 123.45 ou 9.30 são válidos, mas 123.456 não seria. No MySQL, NUMERIC é sinônimo de DECIMAL , e ambos funcionam da mesma forma.

2.8.4. Coluna ativo : BOOLEAN DEFAULT TRUE

- **Ativo:** Nome da coluna, para indicar se o aluno está ativo ou não.
- **BOOLEAN:** No MySQL, BOOLEAN (ou BOOL) é um apelido para TINYINT(1). Internamente, o banco armazena 0 para falso e 1 para verdadeiro. Isso é perfeito para guardar valores simples de verdadeiro/falso.
- **DEFAULT TRUE:** Esta propriedade define um valor padrão para a coluna. Se você não informar um valor para ativo ao inserir um novo registro, o MySQL automaticamente definirá como TRUE (ou 1). Isso é útil para que todos os novos alunos sejam considerados ativos por padrão, a menos que você especifique o contrário.

2.8.5. Coluna criado_em : TIMESTAMP DEFAULT CURRENT_TIMESTAMP

- **criado_em:** Nome da coluna, para registrar a data e hora de criação do registro.

- **TIMESTAMP:** Este tipo de dado armazena uma marca de tempo (data e hora) com precisão até segundos. Ele é automaticamente convertido para UTC (Tempo Universal Coordenado) internamente e, ao ser consultado, é convertido para o fuso horário da sua sessão. É ideal para registrar quando um registro foi criado ou atualizado.
- **DEFAULT CURRENT_TIMESTAMP:** Esta propriedade garante que, se você não informar manualmente o valor para esta coluna, o MySQL automaticamente preencherá com a data e hora atuais do momento da inserção do registro. Isso é muito útil para manter um histórico de quando os dados foram adicionados.

Note que a última coluna (criado_em) não termina com vírgula, pois é a última definição antes de fechar o parêntese da tabela.

2.9. Executando o Comando de Criação da Tabela

Com o comando CREATE TABLE pronto e finalizado, precisamos executá-lo para que a tabela seja de fato criada no seu banco de dados. Antes de executar, é crucial definir em qual banco de dados a tabela será criada. Existem duas formas principais de fazer isso:

1. **Configurando o Banco de Dados Padrão (Recomendado para o Curso):**

- No painel SCHEMAS do MySQL Workbench, clique com o botão direito do mouse em cima do seu banco de dados (curso).

- Selecione a opção Set as Default Schema. Ao ativar, o nome do Schema ficará em negrito, indicando que agora ele é o banco de dados padrão para a execução das suas queries na Query 1.
- Esta é a configuração que será utilizada no curso, pois simplifica a escrita dos comandos.

1. **Alterando o Comando (Para Uso Avançado ou em Outros Contextos):**

- Você também pode especificar o banco de dados diretamente no comando SQL, usando a sintaxe USE nome_do_banco; antes do CREATE TABLE , ou prefixando o nome da tabela com o nome do banco de dados, como CREATE TABLE curso. alunos (...) . Nesse caso, mesmo que outro Schema esteja selecionado como padrão, a tabela será criada dentro de curso porque o nome do Schema está explicitamente indicado.

- Por enquanto, vamos manter o comando padrão, com o nosso banco de dados definido como padrão.

2.9.1. Executando o Comando:

Após definir o banco de dados padrão, clique no ícone de um raio (geralmente localizado acima da área da Query 1) para executar o comando. Se tudo estiver correto, você verá uma mensagem de sucesso na área e será criada com as colunas e tipos de dados solicitados.

2.10. Exercício Prático

Para consolidar o que aprendemos, vamos a um desafio prático! Imagine que uma biblioteca comunitária de uma cidade recrutou você para modelar o banco de dados usado no cadastro de seu acervo. Sua primeira tarefa é criar uma tabela de livros que contenha as seguintes informações:

- Número de identificação do livro
- Nome do livro
- Preço do livro
- Indica se a biblioteca possui exemplares (status sim/não)
- Data em que o livro foi registrado no acervo

2.10.1. Instruções:

- Crie um novo banco de dados chamado `biblioteca_comunitaria` utilizando:

sql
```sql
CREATE DATABASE biblioteca_comunitaria;
```

Não se esqueça do ponto e vírgula (;) para finalizar o comando.

- Execute esse comando para criar o banco de dados.
- Atualize o painel SCHEMAS no MySQL Workbench para visualizar o novo banco.
- Clique com o botão direito sobre biblioteca_comunitaria selecione **Set as Default Schema** para defini-lo como padrão.

- Dentro desse Schema padrão, crie a tabela `livros` usando os tipos de dados mais adequados para cada coluna solicitada. Use restrições como `NOT NULL`, `PRIMARY KEY`, `AUTO_INCREMENT`, `DEFAULT` quando fizerem sentido.

3. CRUD Básico

Olá! Seja muito bem-vindo(a) à sua terceira aula de Banco de Dados com MySQL. Estamos animados para continuar nossa jornada no mundo dos dados e aprofundar seus conhecimentos. Prepare-se para aprender conceitos fundamentais que são a base de qualquer aplicação que lida com informações!

3.1. O que é CRUD?

Nesta aula, vamos desvendar um dos pilares do gerenciamento de dados: o CRUD. Mas o que significa essa sigla? CRUD é um acrônimo formado pelas letras iniciais de quatro operações básicas e essenciais que podemos realizar com os dados dentro de um banco de dados:

- **C - Create** (Criar)
- **R - Read** (Ler)
- **U - Update** (Atualizar)
- **D - Delete** (Excluir)

Essas operações são a espinha dorsal de qualquer sistema que manipula informações. Pense em qualquer aplicativo que você usa diariamente: uma rede social, um aplicativo de banco, uma loja online. Todos eles dependem do CRUD para funcionar. Quando você cria um perfil, lê suas mensagens, atualiza suas informações ou exclui uma postagem, você está realizando operações CRUD!

3.1.1. A Importância do CRUD

Por que o CRUD é tão importante? Simplesmente porque ele representa o ciclo de vida completo dos dados em um sistema. Desde o momento em que uma informação é gerada até o momento em que ela é removida, ela passa por essas quatro fases. Seja um site, um aplicativo móvel ou um sistema empresarial complexo, sempre haverá a necessidade de:

- **Criar** novos registros (ex: cadastrar um novo usuário).
- **Ler** dados existentes (ex: exibir a lista de produtos).
- **Atualizar** informações (ex: mudar o endereço de um cliente).
- **Excluir** dados que não são mais necessários (ex: remover um item do carrinho de compras).

Compreender o CRUD não é apenas sobre memorizar comandos, mas sim sobre entender a lógica por trás de como os dados são gerenciados e manipulados em qualquer sistema. É um conhecimento fundamental que você levará para qualquer tecnologia de banco de dados que venha a aprender no futuro.

Agora que sabemos o que é CRUD e por que ele é tão vital, vamos explorar cada uma dessas operações em detalhes, com exemplos práticos no MySQL.

3.2. Os Comandos do CRUD na Prática

Vamos agora mergulhar nos comandos SQL que nos permitem executar as operações CRUD no MySQL. É importante lembrar que, embora os exemplos sejam focados no MySQL, os conceitos

são universais para a maioria dos bancos de dados relacionais.

3.2.1. C - Create (Criar)

A operação Create é o primeiro passo no ciclo de vida dos dados. É o ato de adicionar novos registros ao seu banco de dados. Pense em um formulário de cadastro de usuários em um site: quando você preenche seus dados e clica em 'Cadastrar', você está realizando uma operação de Create.

3.2.2. Comando INSERT INTO

No MySQL, o comando utilizado para criar novos registros é o INSERT INTO. Ele nos permite especificar em qual tabela queremos inserir os dados e quais valores serão atribuídos a cada coluna.

3.2.2.0.1. Sintaxe Básica:

```sql
INSERT INTO nome_da_tabela (coluna1,
coluna2, coluna3, ...)
VALUES (valor1, valor2, valor3, ...);
```

- **nome_da_tabela:** O nome da tabela onde você deseja inserir o novo registro.
- **(coluna1, coluna2, coluna3, ...):** Uma lista das colunas nas quais você deseja inserir dados. É uma boa prática sempre especificar as colunas, mesmo que você esteja inserindo dados em todas elas, para evitar erros e tornar seu código mais legível.
- **(valor1, valor2, valor3, ...):** Os valores correspondentes a cada coluna, na mesma ordem em que as colunas foram listadas. Valores de texto devem estar entre aspas simples (' '), enquanto números e booleanos (TRUE/FALSE) não precisam.

3.2.3. Exemplo Prático e Explicação Detalhada

Vamos usar o exemplo do nosso roteiro para entender melhor:

3.2.3.0.1. Comando:

```sql
INSERT INTO clientes (nome, email) VALUES
('Ana Souza', 'ana@email.com');
```

3.2.4. Explicação:

Neste comando, estamos dizendo ao MySQL:

- **INSERT INTO clientes:** "Quero inserir um novo registro na tabela chamada clientes."
- **(nome, email):** "Os dados que vou fornecer são para as colunas nome e email."
- **VALUES ('Ana Souza', 'ana@email.com'):** "Os valores para essas colunas serão 'Ana Souza' para nome e 'ana@email.com' para email."

3.2.5. Um pouco mais a fundo:

Imagine que a tabela clientes também tivesse uma coluna id (que é preenchida automaticamente) e uma coluna data_cadastro. Ao especificar apenas nome e e-mail no INSERT INTO, estamos informando ao banco que queremos preencher apenas essas duas colunas com os valores fornecidos. As outras colunas, como id e data_cadastro, serão tratadas de acordo com suas configurações (auto-incremento, valor padrão, etc.).

3.2.6. História para ilustrar:

Pense em uma biblioteca. Quando um novo livro é adquirido, ele precisa ser catalogado. O INSERT INTO é como o bibliotecário preenchendo a ficha de um novo livro: ele anota o título, o autor, o ano de publicação e onde o livro

será guardado. Cada nova ficha é um novo registro, e o INSERT INTO garante que todas as informações essenciais sejam registradas corretamente no sistema da biblioteca (o banco de dados).

Ao final desta operação, um novo registro é adicionado à sua tabela, pronto para ser consultado, atualizado ou, eventualmente, excluído.

3.2.7. R - Read (Ler)

A operação Read é, provavelmente, a mais utilizada em qualquer sistema. Ela nos permite consultar, buscar e visualizar os dados que já estão armazenados no banco de dados. Sempre que você abre um aplicativo e vê informações (suas mensagens, seus produtos favoritos, seu histórico de compras), você está realizando uma operação de Read.

3.2.8. Comando SELECT

No MySQL, o comando principal para ler dados é o SELECT. Ele é extremamente versátil e permite que você especifique exatamente quais colunas deseja ver e de quais tabelas.

3.2.9. Sintaxe Básica:

```sql
SELECT coluna1, coluna2, ...
FROM nome_da_tabela;
```

Ou para selecionar todas as colunas:

```sql
SELECT *
FROM nome_da_tabela;
```

- coluna1, coluna2, ...: As colunas específicas que você deseja visualizar. Se você quiser ver todas as colunas, use o asterisco (*).
- nome_da_tabela: A tabela de onde você deseja buscar os dados.

3.2.10. Exemplo Prático e Explicação Detalhada

Vamos analisar o exemplo do roteiro:

Comando:

```sql
SELECT * FROM alunos;
```

Explicação:

Este comando é bastante direto:

- SELECT *: "Quero selecionar todas as colunas (campos)."
- FROM alunos: "Da tabela chamada alunos."

Ou seja, você está pedindo ao MySQL: "Mostre-me todas as colunas e todos os registros da tabela alunos."

Dica Importante:

Embora SELECT * seja ótimo para testes e para ter uma visão geral dos dados, em aplicações reais, é uma boa prática selecionar apenas as colunas que você realmente precisa. Isso otimiza o desempenho do banco de dados e reduz a quantidade de dados transferidos. Por exemplo, se você quisesse ver apenas o nome e a nota dos alunos, o comando seria:

```sql
SELECT nome, nota FROM alunos;
```

3.2.11. História para ilustrar:

Voltando à nossa biblioteca, o SELECT é como o bibliotecário procurando por um livro específico ou listando todos os livros de um determinado autor. Se ele usa SELECT * FROM livros;, é como se ele pegasse o catálogo inteiro e lesse todas as informações de todos os livros. Mas se ele usa SELECT titulo, autor FROM livros

WHERE gênero = 'Ficção';, ele está sendo mais específico, buscando apenas o título e o autor dos livros de ficção, sem se preocupar com a data de publicação ou o número de prateleira naquele momento.

Após executar um comando SELECT, o banco de dados retorna um conjunto de resultados (uma ou mais linhas) que correspondem à sua consulta, permitindo que você visualize e utilize os dados conforme necessário.

3.2.12. U - Update (Atualizar)

A operação Update é utilizada quando precisamos modificar ou corrigir dados que já existem no banco de dados. Situações comuns incluem a atualização de um endereço de e-mail, a mudança de status de um pedido, ou a correção de um erro de digitação em um nome.

3.2.13. Comando UPDATE

No MySQL, o comando para atualizar registros é o UPDATE. Ele permite que você defina novos valores para uma ou mais colunas em registros específicos.

Sintaxe Básica:

```
UPDATE nome_da_tabela
SET coluna1 = novo_valor1, coluna2 =
novo_valor2, ...
WHERE condicao;
```

- nome_da_tabela: A tabela onde você deseja atualizar os registros.
- SET coluna1 = novo_valor1, ...: Define os novos valores para as colunas especificadas.
- WHERE condição: Esta é a parte mais crítica do comando UPDATE. A condição WHERE especifica quais registros serão atualizados. Se você omitir a cláusula WHERE, todos os registros da tabela serão atualizados, o que pode causar perda de dados irreversível e indesejada.

3.2.14. Exemplo Prático e Explicação Detalhada

Vamos analisar o exemplo do roteiro:

Comando:

```sql
UPDATE alunos SET ativo = FALSE WHERE id =
1;
```

Explicação:

Este comando está realizando a seguinte ação:

- UPDATE alunos: "Quero modificar um registro na tabela alunos."
- SET ativo = FALSE: "Defina o valor da coluna ativo para FALSE (falso)."
- WHERE id = 1: "Apenas para o registro onde o id é igual a 1."

Em outras palavras, estamos marcando o aluno com id = 1 como inativo no banco de dados. Isso é útil, por exemplo, se um aluno trancou a matrícula ou concluiu o curso.

3.2.15. A Importância do WHERE:

Como mencionado, a cláusula WHERE é vital. Se o comando fosse UPDATE alunos SET ativo = FALSE; (sem o WHERE), todos os alunos da tabela seriam marcados como inativos, o que raramente é o desejado. Por isso, sempre use o WHERE com um identificador único (como id) para garantir que apenas o registro correto seja alterado.

História para ilustrar:

Imagine que você tem uma lista de contatos no seu celular. Um amigo mudou de número de telefone. A operação UPDATE é como você abrindo o contato dele e editando o número antigo para o novo. Você não mudaria o número de todos os seus contatos, certo? Você especifica exatamente qual contato quer atualizar. O WHERE é o que te permite encontrar o contato

certo (pelo nome, por exemplo) antes de fazer a alteração.

Após a execução bem-sucedida de um comando UPDATE, os dados no banco de dados são modificados de acordo com as instruções, refletindo as informações mais recentes.

3.2.16. D - Delete (Excluir)

A operação Delete é a última do ciclo de vida dos dados. Ela é utilizada quando precisamos remover um registro do banco de dados. Isso pode acontecer quando um usuário exclui sua conta, um produto é descontinuado, ou uma informação se torna obsoleta.

3.2.17. Comando DELETE

No MySQL, o comando para excluir registros é o DELETE. Assim como o UPDATE, ele depende crucialmente da cláusula WHERE para especificar quais registros devem ser removidos.

Sintaxe Básica:

```sql
DELETE FROM nome_da_tabela
WHERE condicao;
```

- nome_da_tabela: A tabela de onde você deseja excluir os registros.
- WHERE condicao: A condição que especifica quais registros serão excluídos. Se você omitir a cláusula WHERE, todos os registros da tabela serão excluídos, o que é uma ação extremamente perigosa e geralmente indesejada.

Exemplo Prático e Explicação Detalhada

Vamos analisar o exemplo do roteiro:

Comando:

```sql
DELETE FROM alunos WHERE id = 1;
```

Explicação:

Este comando está realizando a seguinte ação:

- DELETE FROM alunos: "Quero apagar um ou mais registros da tabela alunos.
- WHERE id = 1: "Apenas o registro onde o id é igual a 1."

Em resumo, estamos removendo permanentemente o aluno com id = 1 do nosso banco de dados.

A Importância do WHERE (novamente!):

Assim como no UPDATE, a cláusula WHERE é fundamental no DELETE. Um comando DELETE FROM alunos; (sem o WHERE) apagaria todos os alunos da tabela de uma vez, o que seria um erro catastrófico na maioria dos casos. Sempre tenha certeza de que sua cláusula WHERE está correta antes de executar um comando DELETE.

História para ilustrar:

Pense em uma caixa de e-mails. Quando você seleciona um e-mail e clica em "Excluir", você está realizando uma operação DELETE. Você não clica em um botão que apaga todos os seus e-mails de uma vez, a menos que seja essa a sua intenção. Você especifica qual e-mail quer remover. O WHERE é o que garante que apenas o e-mail selecionado seja movido para a lixeira.

Após a execução de um comando DELETE, os registros que correspondem à condição WHERE são permanentemente removidos do banco de dados. É uma ação poderosa e deve ser usada com cuidado.

3.2.18. MySQL Workbench: Preparando o Ambiente

Para colocar em prática tudo o que aprendemos sobre CRUD, utilizaremos o MySQL Workbench, uma ferramenta visual que nos permite interagir com o banco de dados de forma intuitiva. É o ambiente onde você escreverá e executará seus comandos SQL.

Acessando o Serviço Local

Ao abrir o MySQL Workbench, você verá uma tela inicial com suas conexões. Para acessar o nosso serviço local de MySQL, siga estes passos:

1. Com o MySQL Workbench aberto, procure pela seção "MySQL Connections".
2. Clique na conexão "local mysql" (ou o nome que você configurou para sua conexão local).

Este é o seu portal para o banco de dados! Acostume-se com ele, pois será seu principal ambiente de trabalho.

Entendendo a Opção "Save password in vault"

Antes de inserir sua senha para acessar a conexão, você pode ter notado uma opção chamada "Save password in vault". Vamos entender o que ela faz:

- **Significado:** Traduzida para o português, significa "Salvar a senha no cofre".
- **Funcionalidade:** Quando você marca essa caixinha, o MySQL Workbench armazena a senha que você digitou de forma segura no seu computador. Ele utiliza um sistema de armazenamento protegido, conhecido como "vault" (cofre).
- **Benefício:** Na prática, isso significa que você não precisará digitar a senha toda vez que abrir o Workbench e quiser se conectar ao seu MySQL local. O Workbench lembrará da senha para aquela conexão específica, tornando seu acesso mais rápido e conveniente.

Importante: Embora seja conveniente, certifique-se de que seu ambiente de trabalho é seguro antes de usar essa opção, especialmente em computadores compartilhados.

Após ativar essa opção e inserir sua senha (por exemplo, 123456), clique em "OK". Parabéns! Você estará dentro do seu serviço MySQL local, pronto para começar a executar os comandos CRUD em seu banco de dados.

3.2.19. Prática Guiada: CRUD Completo

Vamos realizar uma prática guiada completa, aplicando todas as operações em nossa tabela alunos no MySQL Workbench. Esta seção simula um cenário real de gerenciamento de dados.

Nossa tabela alunos possui as colunas nome (texto), nota (decimal) e ativo (booleano), além de colunas que são preenchidas automaticamente, como id e criacao_em.

3.2.20. Criando um Novo Registro (CREATE)

Cenário: Um novo aluno, "Anderson da Silva", acabou de se matricular. Ele ainda não tem nota e seu status inicial é inativo.

Comando:

```sql
INSERT INTO alunos (nome, nota, ativo)
VALUES (\"Anderson da Silva\", 0, FALSE);
```

Explicação: Estamos inserindo um novo registro na tabela alunos. O nome é "Anderson da Silva", a nota inicial é 0, e o ativo é FALSE, indicando que ele está inativo no momento.

3.2.21. Consultando o Registro Criado (READ)

Cenário: Precisamos verificar se o registro de "Anderson da Silva" foi inserido corretamente e qual o id que o banco de dados atribuiu a ele.

Comando:

```sql
SELECT * FROM alunos;
```

Explicação: Este comando nos permite visualizar todos os registros e todas as colunas da tabela alunos. Você deverá ver o novo registro de "Anderson da Silva" com seu id (provavelmente 2, se for o segundo registro inserido) e a criacao_em preenchida automaticamente.

3.2.22. Atualizando o Registro (UPDATE)

Cenário: "Anderson da Silva" começou a frequentar as aulas e agora precisa ser ativado no sistema.

Comando:

```sql
UPDATE alunos SET ativo = TRUE WHERE id = 2;
```

Explicação: Estamos atualizando o status do aluno com id = 2 (Anderson da Silva) para ativo = TRUE. Lembre-se de usar o id correto do aluno que você criou.

3.2.23. Confirmando a Atualização (READ)

Cenário: Queremos ter certeza de que a atualização do status de "Anderson da Silva" foi efetivada.

Comando:

```sql
SELECT nome, ativo FROM alunos WHERE id = 2;
```

Explicação: Consultamos apenas as colunas nome e ativo para o aluno com id = 2. O resultado deve mostrar "Anderson da Silva" com ativo como TRUE (ou 1).

Excluindo o Registro (DELETE)

Cenário: "Anderson da Silva" precisou se transferir para outra escola e seu registro deve ser removido do nosso sistema.

Comando:

```sql
DELETE FROM alunos WHERE id = 2;
```

Explicação: Este comando remove o registro do aluno com id = 2 da tabela alunos. Cuidado: Esta ação é irreversível.

3.2.24. Confirmando a Exclusão (READ)

Cenário: Precisamos verificar se o registro de "Anderson da Silva" foi realmente excluído.

Comando

```sql
SELECT * FROM alunos;
```

Explicação: Ao executar este comando, você notará que o registro de "Anderson da Silva" não aparece mais na lista, confirmando sua exclusão.

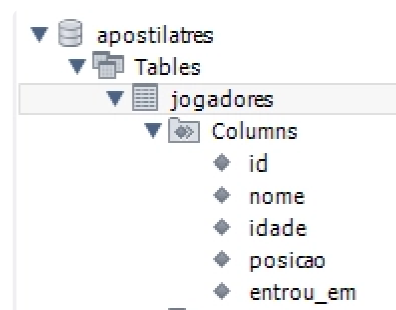
Parabéns! Você acaba de realizar um ciclo completo de CRUD na prática. Isso demonstra o poder e a importância dessas quatro operações no gerenciamento de dados.

3.3. Exercícios práticos

Como exercício prático desta apostila, vamos realizar a utilização do CRUD. Primeiro crie um banco de dados chamado "apostilatres", depois crie uma tabela jogadores com as seguintes colunas:

- id (inteiro, chave primária, auto incremento)
- nome (texto de até 100 caracteres, obrigatório)
- idade (inteiro, obrigatório)
- posicao (texto de até 100 caracteres)
- entrou_em (registro de data e hora com valor padrão automático)

Deverá ficar da seguinte forma abaixo:



Agora que você criou a tabela jogadores, insira um novo jogador com as seguintes informações:

- Nome: João Oliveira
- Idade: 24
- Posição: Lateral
- Data e hora de entrada: deixe o banco preencher automaticamente

Após criar, realize uma consulta e visualize se consta conforme o abaixo:

	id	nome	idade	posicao	entrou_em
▶	1	João Oliveira	24	Lateral	2025-08-04 17:10:46
*	NULL	NULL	NULL	NULL	NULL

Agora vamos atualizar os dados do jogador que você acabou de inserir. Atualize o jogador João Oliveira para:

- Idade: 25
- Posição: Meia

Depois de criar consulte se consta atualizado o registro conforme abaixo:

	id	nome	idade	posicao	entrou_em
▶	1	João Oliveira	25	Meia	2025-08-04 17:10:46
*	NULL	NULL	NULL	NULL	NULL

Agora, exclua o jogador e depois consulte para verificar se foi excluído. Deverá ficar em branco, sem registro conforme abaixo:

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap

	id	nome	idade	posicao	entrou_em
*	NULL	NULL	NULL	NULL	NULL

Result Grid
Form Editor
Field Types

3.4. Conclusão

Chegamos ao final da nossa aula sobre CRUD em MySQL! Esperamos que você tenha compreendido a importância dessas quatro operações fundamentais: Criar, Ler, Atualizar e Excluir. Elas são a base para qualquer interação com bancos de dados e, ao dominar esses conceitos, você estará muito mais preparado para desenvolver aplicações robustas e eficientes.

Lembre-se que a prática leva à perfeição. Continue experimentando os comandos no MySQL Workbench, crie suas próprias tabelas, insira dados, consulte-os de diferentes formas, atualize informações e, com cautela, pratique a exclusão de registros. Quanto mais você praticar, mais natural se tornará o uso do SQL e dos conceitos de banco de dados.

Anotações

Bem-vindo(a) à nossa quarta aula de banco de dados com MySQL! Nesta apostila, vamos aprofundar nossos conhecimentos sobre como manipular dados de forma eficiente, utilizando técnicas de filtragem, ordenação e limitação. Prepare-se para desvendar o poder do SQL e otimizar suas consultas!

4.1. Filtragem de Dados com WHERE

A filtragem de dados é uma das operações mais poderosas em SQL, permitindo que você selecione apenas os registros que realmente importam em um vasto conjunto de informações. É como ter um superpoder para encontrar exatamente o que você precisa em um piscar de olhos. O comando WHERE é a chave para desbloquear esse poder.

Imagine que seu banco de dados é uma biblioteca gigantesca, cheia de livros sobre os mais variados assuntos. Se você quer encontrar todos os livros de ficção científica escritos por autores brasileiros, não vai folhear livro por livro, certo? Você usaria um sistema de busca, um filtro. No MySQL, o WHERE faz exatamente isso: ele atua como um filtro, permitindo que você especifique critérios para que apenas os dados que correspondem a esses critérios sejam exibidos.

4.1.1. Comparadores Simples

Os comparadores simples são a base da filtragem. Eles permitem que você compare valores em suas colunas com um valor específico. Pense neles como as perguntas mais básicas que você faria a um conjunto de dados:

- **= (Igual a):** Quer encontrar todos os alunos que tiraram exatamente 9.20 na prova? Use o =. É como perguntar: "Quem tirou a nota exata que eu quero?".
- **> (Maior que):** E se você quiser ver quem tirou mais de 8.0? O > é o seu comparador. Ele seleciona todos os valores estritamente maiores que o especificado.
- **< (Menor que):** Para identificar quem precisa de um pouco mais de atenção, você pode usar o < para encontrar notas menores que um determinado valor.
- **<= (Menor ou igual a):** Se a sua condição inclui o valor limite, use o <=. Por exemplo, notas de 8.0 para baixo.
- **>= (Maior ou igual a):** Da mesma forma, para incluir o valor limite quando a condição é maior, use o >=. Notas de 8.0 para cima.
- **!= ou <> (Diferente de):** Quer ver todos os alunos, exceto aqueles que tiraram uma nota específica? O != (ou <>) faz isso. Ele retorna todos os registros que não correspondem ao valor especificado.

É importante notar que, no MySQL, tanto != quanto <> significam a mesma coisa: "não é igual a". A escolha entre um e outro é mais uma questão de preferência pessoal ou padrão da equipe.

4.1.2. Intervalo de Valores com BETWEEN...AND

Às vezes, você precisa filtrar dados que se encontram dentro de um determinado intervalo. Em vez de usar múltiplos comparadores (>= e <=), o BETWEEN...AND oferece uma maneira mais elegante e legível de fazer isso.

```sql
SELECT * FROM alunos WHERE nota BETWEEN 8.00
AND 9.50;
```

Este comando seleciona todos os alunos cuja nota está entre 8.00 e 9.50, **incluindo** os próprios valores 8.00 e 9.50. É como dizer: "Quero todos

os alunos que tiraram notas de 8.00 a 9.50, inclusive as extremidades".

História: A universidade estava oferecendo um programa de intercâmbio para alunos com notas entre 8.00 e 9.50. A cláusula BETWEEN...AND simplificou a seleção dos candidatos ideais, garantindo que apenas os alunos dentro dessa faixa fossem considerados.

E se você quiser o oposto? Ou seja, todos os valores que não estão dentro de um intervalo? Para isso, usamos o NOT BETWEEN:

```sql
SELECT * FROM alunos WHERE nota NOT BETWEEN
8.00 AND 9.50;
```

Este comando retornaria todos os alunos com nota abaixo de 8.00 ou acima de 9.50. É o inverso do BETWEEN...AND.

4.1.3. Lista de Valores com IN e NOT IN

Quando você precisa filtrar registros com base em uma lista específica de valores, os operadores IN e NOT IN são extremamente úteis. Eles evitam que você precise escrever várias condições OR.

```sql
SELECT * FROM alunos WHERE nome IN ('Ana
Souza', 'Fernanda Lima');
```

Este comando seleciona todos os alunos cujo nome seja exatamente "Ana Souza" ou "Fernanda Lima". Pense nisso como ter uma lista de convidados para uma festa e querer saber quem, da sua tabela de alunos, está nessa lista. O IN faz essa verificação de forma eficiente.

História: A diretora da escola estava organizando uma reunião com os pais de alguns alunos específicos para discutir o desempenho acadêmico. Em vez de procurar um por um, ela usou o IN para gerar rapidamente a lista de contatos de "Ana Souza" e "Fernanda Lima", agilizando o processo.

Para o cenário oposto, onde você quer excluir registros que estão em uma lista de valores, usamos o NOT IN:

```sql
SELECT * FROM alunos WHERE nota NOT IN
(7.50, 9.20);
```

Este comando seleciona todos os alunos cuja nota seja diferente de 7.50 e também diferente de 9.20. É como dizer: "Quero todos os alunos, exceto aqueles que tiraram 7.50 ou 9.20".

História: O professor de educação física estava montando os times para um torneio e queria que todos os alunos participassem, exceto aqueles que já estavam no time de basquete (que tinham notas 7.50 e 9.20 em outra disciplina). O NOT IN ajudou a garantir que a seleção fosse justa e que novos talentos fossem descobertos.

4.1.4. Padrões de Texto com LIKE

Filtrar por padrões de texto é essencial quando você não sabe o valor exato de uma string, mas tem uma ideia de como ela começa, termina ou o que ela contém. O operador LIKE é a ferramenta para isso, e ele usa caracteres curinga:

- **% (Porcentagem):** Representa zero ou mais caracteres. É o curinga mais comum.
- **_ (Sublinhado):** Representa um único caractere.

Vamos ver como usar o %:

```sql
SELECT * FROM alunos WHERE nome LIKE 'A%';
```

Este comando retorna todos os alunos cujos nomes começam com a letra "A". Por exemplo, "Ana", "André", "Alice", "Antonella" apareceriam na lista. É como procurar em um catálogo telefônico por todos os nomes que começam com uma letra específica.

História: A secretaria da escola precisava gerar uma lista de todos os alunos cujos nomes

começavam com a letra 'A' para um evento de boas-vindas. O LIKE 'A%' foi a solução perfeita para essa tarefa, poupando horas de busca manual.

Você também pode usar o % no meio ou no final da string:

sql

```sql
SELECT * FROM alunos WHERE nome LIKE
'%nan%';
```

Este comando retorna os alunos cujo nome contém a sequência "nan" em qualquer parte do texto (no começo, meio ou fim). Por exemplo, "Fernanda" seria um resultado.

História: Um pesquisador de nomes estava interessado em identificar padrões em nomes de alunos. Ele usou o LIKE '%nan%' para encontrar todos os nomes que continham a sequência "nan", revelando uma curiosa prevalência em algumas turmas.

4.1.5. Combinação Lógica com AND, OR e NOT

Para criar filtros mais complexos, você pode combinar múltiplas condições usando operadores lógicos. Eles permitem que você construa consultas que atendam a vários critérios simultaneamente ou alternativamente.

- **AND (E):** O operador AND exige que todas as condições sejam verdadeiras para que um registro seja retornado. É como dizer: "Quero os alunos que atendem à condição A E à condição B".
- **OR (OU):** O operador OR exige que pelo menos uma das condições seja verdadeira. É como dizer: "Quero os alunos que atendem à condição A OU à condição B (ou ambas)".
- **NOT (NÃO):** O operador NOT nega uma condição, ou seja, ele retorna o oposto do que a condição especifica. É como dizer: "Quero os alunos que NÃO atendem a esta condição".

4.1.6. Valores Ausentes com IS NULL e IS NOT NULL

Em bancos de dados, é comum encontrar campos sem valor, ou seja, NULL. É importante saber como lidar com eles, pois comparadores comuns (=, !=) não funcionam com NULL da mesma forma que funcionam com outros valores. Para isso, usamos IS NULL e IS NOT NULL.

- **IS NULL:** Seleciona registros onde um campo não tem nenhum valor definido (está vazio).
- **IS NOT NULL:** Seleciona registros onde um campo tem algum valor definido (não está vazio).

4.2. Ordenação de Dados com ORDER BY

Depois de filtrar os dados para obter apenas o que interessa, muitas vezes você vai querer organizá-los de uma forma específica. É aqui que entra o ORDER BY, a cláusula que permite ordenar o resultado da sua consulta. Pense em organizar uma pilha de documentos: você pode querer ordená-los por data, por nome, ou por qualquer outro critério. O ORDER BY faz exatamente isso com seus dados no MySQL.

4.2.1. Ordenação Básica: ASC e DESC

A forma mais comum de ordenar dados é em ordem crescente ou decrescente. O ORDER BY oferece dois modificadores para isso:

- **ASC (Ascendente):** Ordena os resultados do menor para o maior. Para números, é do menor para o maior. Para textos, é em ordem alfabética (A-Z). Se você não especificar ASC ou DESC, o ASC é o padrão.
- **DESC (Descendente):** Ordena os resultados do maior para o menor. Para números, é do maior para o menor. Para textos, é em ordem alfabética inversa (Z-A).

Você também pode ordenar por múltiplas colunas. O MySQL ordenará primeiro pela primeira coluna especificada, e se houver valores iguais, usará a segunda coluna para desempate, e assim por diante.

sql

```sql
SELECT * FROM alunos ORDER BY ativo DESC,
nota ASC;
```

Neste exemplo, ele ordena primeiro por ativo (verdadeiros primeiro, pois DESC para booleanos geralmente coloca TRUE antes de FALSE), depois por nota em ordem crescente. É uma forma eficaz de combinar critérios de ordenação.

4.2.2. Tratamento de Valores Nulos na Ordenação

Valores NULL (ausentes) podem se comportar de forma diferente na ordenação, dependendo do sistema de banco de dados e da configuração. No MySQL, por padrão, NULL é tratado como o menor valor em uma ordenação ASC e o maior valor em uma ordenação DESC.

No entanto, você pode controlar explicitamente onde os valores NULL aparecem na sua ordenação usando expressões com IS NULL ou IS NOT NULL dentro do ORDER BY.

sql

```sql
SELECT * FROM alunos ORDER BY (nome IS
NULL), nome ASC;
```

Utilizando (nome IS NULL) dentro do ORDER BY, você está criando uma coluna temporária booleana. Nome IS NULL retorna TRUE (ou 1) se o nome for NULL e FALSE (ou 0) se não for. Como FALSE (0) é menor que TRUE (1), os nomes não nulos virão primeiro (ordenados por nome ASC), seguidos pelos nulos.

História: A equipe de RH de uma empresa precisava de uma lista de funcionários, mas queria que aqueles com informações de contato completas aparecessem primeiro. Ao ordenar por (email IS NULL), nome ASC, eles garantiram que os funcionários com e-mail preenchido fossem listados antes dos que tinham o campo vazio, facilitando o contato.

O inverso também é possível com IS NOT NULL se você quiser que os nulos apareçam primeiro.

4.2.3. Ordenação Sensível ou Insensível a Maiúsculas/Minúsculas

Ao ordenar strings, o MySQL pode ser sensível ou insensível a maiúsculas e minúsculas, dependendo da collation (conjunto de regras de comparação de caracteres) da coluna ou do banco de dados. Por padrão, muitas collations no MySQL são insensíveis a maiúsculas/minúsculas (por exemplo, utf8_general_ci, onde ci significa case-insensitive).

Se você precisa de uma ordenação que diferencie maiúsculas de minúsculas, pode usar o operador BINARY.

sql

```sql
SELECT * FROM alunos ORDER BY BINARY nome;
```

Este comando faz com que o banco ordene os nomes de maneira sensível a maiúsculas e minúsculas. Isso significa que "Ana" e "ana" serão tratados como valores diferentes, e a ordem pode ser "Ana", "André", "ana", "antonio", dependendo da representação binária dos caracteres.

História: Um sistema de registro de senhas precisava de uma ordenação estritamente sensível a maiúsculas e minúsculas para fins de segurança e auditoria. O uso do BINARY na ordenação garantiu que as senhas fossem tratadas com a precisão necessária, sem ignorar as diferenças entre letras maiúsculas e minúsculas.

Com o ORDER BY, você tem controle total sobre como seus resultados são apresentados, tornando a análise e a visualização dos dados muito mais eficazes.

4.3. Limitação de Dados com LIMIT

Para implementar a paginação de forma eficaz, você frequentemente precisará não apenas limitar o número de resultados, mas também "pular" um certo número de registros iniciais. É para isso que serve o OFFSET.

O OFFSET especifica a partir de qual linha (contando a partir do zero) o MySQL deve começar a retornar os resultados. Por exemplo, se você quer a segunda "página" de resultados, você pularia a primeira página.

```sql
SELECT * FROM alunos ORDER BY nota DESC
LIMIT 3 OFFSET 2;
```

Neste comando de exemplo, o MySQL primeiro ordena os alunos pela nota em ordem decrescente. Em seguida, ele pula os dois primeiros resultados e retorna os próximos três. Se a lista completa fosse:

1. Aluno A (nota 10)
2. Aluno B (nota 9.5)
3. Aluno C (nota 9.0)
4. Aluno D (nota 8.5)
5. Aluno E (nota 8.0)

Com LIMIT 3 OFFSET 2, os resultados seriam: Aluno C, Aluno D, Aluno E.

História: Um aplicativo de notícias precisava exibir os artigos mais recentes em blocos de 5. Para a segunda página, eles usaram LIMIT 5 OFFSET 5, garantindo que os próximos 5 artigos fossem carregados sem repetir os da primeira página. Essa combinação de LIMIT e OFFSET é a base de quase toda a paginação de dados na web.

O LIMIT e o OFFSET são ferramentas poderosas para controlar o volume de dados que você recupera, otimizando o desempenho das suas consultas e melhorando a experiência do usuário em aplicações que lidam com grandes conjuntos de dados.

4.4. A Importância da Combinação de Filtragem, Ordenação e Limitação

Agora que exploramos individualmente a filtragem (WHERE), a ordenação (ORDER BY) e a limitação (LIMIT), é crucial entender por que esses três recursos, quando combinados, se tornam ferramentas indispensáveis para qualquer desenvolvedor ou analista de dados. Eles não são apenas comandos isolados; são pilares para a construção de consultas eficientes e a apresentação de dados de forma significativa.

Imagine que você é o gerente de uma grande loja online. Você precisa saber:

1. **Quais produtos foram mais vendidos no último mês (Filtragem)?** Você usaria WHERE para restringir as vendas ao período desejado.
2. **Quais são os 10 produtos mais populares (Ordenação e Limitação)?** Você ordenaria os resultados por quantidade vendida (ORDER BY DESC) e pegaria apenas os 10 primeiros (LIMIT 10).
3. **Quais clientes compraram mais de R$ 500,00 e estão localizados em São Paulo, ordenados pelo valor total da compra, e quero ver apenas os 5 primeiros (Combinação)?** Aqui, você usaria WHERE para filtrar por valor e localização, ORDER BY para ordenar pelo valor da compra, e LIMIT para pegar os 5 primeiros.

Essa combinação permite que você extraia informações precisas e relevantes de um mar de dados. A filtragem (WHERE) ajuda a selecionar somente o que importa, eliminando o ruído. A ordenação (ORDER BY) coloca os dados na sequência desejada, facilitando a leitura e a análise. E a limitação (LIMIT) evita a sobrecarga de informação, tornando a consulta mais rápida e os resultados mais fáceis de lidar.

Esses três recursos combinados são usados o tempo todo em sistemas reais: em sites de e-commerce para exibir produtos relevantes, em relatórios gerenciais para apresentar insights de negócios, em painéis de controle para monitorar métricas em tempo real, e em qualquer aplicação

que precise de buscas rápidas, precisas e organizadas.

História: Uma startup de tecnologia estava desenvolvendo um novo painel de controle para seus clientes. Eles perceberam que as consultas estavam lentas e os usuários ficavam perdidos com a quantidade de dados. Ao implementar a combinação de WHERE, ORDER BY e LIMIT, eles conseguiram criar um painel que carregava rapidamente, mostrava apenas as informações mais relevantes e permitia que os usuários navegassem pelos dados de forma intuitiva. O sucesso do painel foi um reflexo direto da aplicação inteligente desses três conceitos.

Dominar a arte de combinar esses comandos é o que diferencia um usuário básico de SQL de um profissional capaz de extrair o máximo valor de um banco de dados. É a base para construir aplicações robustas e eficientes.

4.5. Atividade prática

Antes de começar, crie um banco de dados chamado apostila4, depois crie uma tabela para este banco de dados chamado alunos pratica e insira os registros. Siga os comandos abaixo para realizar estes procedimentos antes de começar a aula prática desta apostila:

1. Criação do banco de dados:

```sql
CREATE DATABASE apostila4;
```

1. Criação de tabela:

```sql
CREATE TABLE apostila4.alunospratica (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    nota DECIMAL(4,2) NOT NULL,
    ativo BOOLEAN NOT NULL,
    email VARCHAR(150)
);
```

1. Inserir registros:

```sql
INSERT INTO apostila4.alunospratica (nome,
nota, ativo, email) VALUES
('Ana Souza', 9.2, TRUE,
'ana.souza@email.com'),
('Carlos Andrade', 8.5, TRUE, NULL),
('Fernanda Lima', 7.8, TRUE,
'fernanda.lima@email.com'),
('João Santos', 6.4, FALSE, NULL),
('Mariana Alves', 9.0, TRUE,
'mariana.alves@email.com'),
('Rafael Nunes', 5.5, FALSE,
'rafael.nunes@email.com'),
('Anderson Costa', 8.7, TRUE, NULL),
('Tatiane Rocha', 7.2, TRUE,
'tatiane.rocha@email.com');
```

Agora com nossa estrutura criada, vamos realizar nosso exercício prático desta aula, realize cada comando para cada solicitação abaixo:

1. Selecionar todos os alunos cuja nota seja maior ou igual a 8.0.
2. Listar apenas os alunos com nota entre 7.5 e 9.0, ordenados por nota em ordem decrescente.
3. Exibir todos os alunos cujo nome contenha a sequência de letras "an", ordenados em ordem alfabética.
4. Mostrar todos os alunos cujo nome seja "Ana Souza" ou "Fernanda Lima".
5. Listar os alunos que não possuem e-mail (campo e-mail é NULL).
6. Selecionar os alunos ativos e com nota acima de 8.5, ordenando primeiro por ativo (DESC) e depois por nota (ASC).
7. Exibir apenas os 5 primeiros resultados da tabela, ordenados pelo nome.
8. Mostrar os alunos cuja nota não esteja entre 6.0 e 8.0.

Dica para resolver, utilize os operadores: =, >, <, >=, <=, != ou <>, BETWEEN, NOT BETWEEN, IN, NOT IN, LIKE, IS NULL, IS NOT NULL, ORDER BY e LIMIT.

Banco de Dados com MySQL

Banco de dados com MySql

5. Agregação e Agrupamento

Aula 5

Olá, seja bem-vindo(a) à nossa quinta aula de banco de dados com MySQL! Nesta sessão, aprofundaremos nosso conhecimento sobre **agregação e agrupamento** em bancos de dados, utilizando o MySQL Workbench como ferramenta prática.

5.1. Agregação e Agrupamento: Conceitos Fundamentais

Agora que revisamos os conceitos básicos, vamos avançar para o cerne desta aula: **agregação e agrupamento.** Estes são pilares essenciais para a análise de dados em qualquer sistema de gerenciamento de banco de dados relacional, incluindo o MySQL.

5.1.1. O que são Agregação e Agrupamento?

Para entender a agregação e o agrupamento, imagine que você tem uma grande quantidade de dados, como uma pilha de recibos de vendas ou uma folha com as notas de todos os alunos de uma escola. Se você precisasse responder a perguntas como "quantos alunos passaram?", "qual a média das notas da turma?" ou "quantas vendas houve por dia?", ler cada item individualmente seria uma tarefa exaustiva e ineficiente.

É exatamente aqui que a **agregação** e o **agrupamento** entram em ação. Eles fornecem uma maneira automática e rápida de resumir esses dados, transformando muitas linhas de informação em resumos úteis e concisos.

- **Agregação:** Refere-se ao ato de calcular um único valor a partir de um conjunto de valores. Pense nisso como "condensar" dados. Por exemplo, contar quantos registros existem *(COUNT)*, somar valores `<i>(SUM)</i>`, calcular a média *(AVG)*, encontrar o maior valor *(MAX)* ou o menor valor *(MIN)*. Essas são as chamadas **funções agregadas**, e o MySQL oferece um conjunto robusto delas para facilitar a sumarização de dados.

- **Agrupamento (GROUP BY):** O agrupamento ocorre quando você divide a tabela em "grupos" lógicos antes de aplicar uma agregação. Em vez de calcular a média de todas as notas da turma, você pode querer calcular a média por turma, por gênero, ou por status (ativo/inativo). O *GROUP BY* organiza as linhas em "baldes" (grupos) que compartilham o mesmo valor em uma ou mais colunas especificadas. Em seguida, as funções agregadas são aplicadas a cada um desses "baldes" separadamente, fornecendo resumos para cada categoria.

5.1.2. Funções Agregadas Básicas

As funções agregadas são ferramentas poderosas para resumir dados. Vamos conhecer as mais comuns e entender o que cada uma faz:

- `COUNT(*)` : Esta função conta o número total de linhas em um conjunto de resultados. É extremamente útil para saber quantos registros existem em uma tabela ou em um grupo específico.

- `SUM(coluna)` : Utilizada para somar os valores de uma coluna numérica. É indispensável para cálculos financeiros, totais de vendas, pontuações acumuladas, entre outros.

- `AVG(coluna)` : Calcula a média aritmética dos valores de uma coluna numérica. Perfeita para determinar a média de notas, preços, idades, etc.

- `MIN(coluna)` : Encontra o menor valor em uma coluna. Útil para identificar o menor preço, a menor nota, a data mais antiga, etc.

- `MAX(coluna)` : Encontra o maior valor em uma coluna. Complementar ao MIN, serve para identificar o maior preço, a maior nota, a data mais recente, etc.

Exemplo Ilustrativo:

Considere a seguinte consulta:

```sql
SELECT COUNT(*) AS total_alunos, AVG(nota)
AS media_geral FROM alunos;
```

Se tivéssemos alunos com notas (9.0, 8.5, 7.5, 9.8, 10.0), o resultado seria:

total_alunos	media_geral
5	8.96

Este exemplo demonstra como `COUNT(*)` nos dá o número de alunos e `AVG(nota)` nos fornece a média geral das notas, transformando múltiplos dados em dois valores resumidos.

5.1.3. Agrupamento com GROUP BY

O `GROUP BY` permite que você obtenha resumos por categoria, o que é fundamental para análises mais detalhadas. Em vez de um resumo geral, você pode ter resumos para subconjuntos específicos dos seus dados.

Exemplo:

Imagine que você quer saber a média das notas separando os alunos que estão `ativo` (status 1) dos que estão `inativo` (status 0). A consulta seria:

```sql
SELECT ativo, COUNT(*) AS qtd, AVG(nota) AS
media FROM alunos GROUP BY ativo;
```

Explicação: O MySQL agrupa todas as linhas onde ativo = 1 e calcula a contagem (`COUNT`) e a média (`AVG`) para esse grupo. Em seguida, faz o mesmo para as linhas onde `ativo = 0`. O resultado seria algo como:

ativo	qtd	media
1	4	8.91
0	1	9.20

Isso nos mostra que, no grupo de alunos ativos, há alunos com uma média de 8.91, enquanto no grupo de inativos, há 1 aluno com média de 9.20. O `GROUP BY` transformou um resumo único em resumos categorizados.

5.1.4. Filtragem: WHERE vs. HAVING

Um ponto crucial na agregação e agrupamento é entender a diferença entre filtrar linhas antes de agrupar e filtrar grupos depois da agregação. Para isso, utilizamos as cláusulas `WHERE` e `HAVING`.

- **WHERE** : Esta cláusula filtra linhas individuais antes que qualquer agregação seja realizada. Use `WHERE` quando quiser excluir registros do cálculo agregado. Ela atua diretamente sobre as colunas da tabela original.
- **HAVING** : Esta cláusula filtra os grupos resultantes depois que a agregação foi feita. Use `HAVING` quando a condição de filtro depende do resultado de uma função agregada (por exemplo, "mostre apenas os grupos cuja média é maior que "). O `HAVING` "vê" os resultados das funções agregadas.

Exemplo Comparativo:

Se quisermos apenas os grupos cuja média de nota seja maior que , usaríamos:

```sql
SELECT ativo, AVG(nota) AS media FROM alunos
GROUP BY ativo HAVING AVG(nota) > 9;
```

Neste caso, o `HAVING` age sobre a média calculada para cada grupo. Se tentássemos colocar `AVG(nota) > 9` na cláusula `WHERE`, o MySQL geraria um erro, pois o `WHERE` não consegue processar funções agregadas; ele só filtra as linhas "brutas" antes do `GROUP BY`.

5.1.5. Comportamento com Valores NULL e COUNT

Um detalhe importante e que pode gerar confusão é como as funções agregadas lidam com valores `NULL` (nulos). Em geral, as funções agregadas ignoram valores `NULL`, ou seja, eles não são contados, somados ou incluídos na média. No entanto, há uma exceção notável com `COUNT`:

- **COUNT(coluna)** : Conta apenas as linhas onde a coluna especificada não é `NULL`. Se uma célula na coluna nota estiver vazia `(NULL)`, ela não será incluída na contagem.

- **COUNT(*)** : Conta todas as linhas em um grupo, independentemente de qualquer coluna conter valores `NULL`. Isso significa que `COUNT(*)` incluirá registros mesmo que a coluna nota esteja `NULL`.

Essa distinção é crucial, pois pode alterar significativamente os resultados da sua consulta, especialmente quando há dados faltando ou incompletos na sua base de dados.

5.1.6. Agrupamento por Múltiplas Colunas

Às vezes, uma única categoria de agrupamento não é suficiente para a análise desejada. O MySQL permite agrupar por duas ou mais colunas simultaneamente, criando subgrupos mais específicos.

Exemplo:

Para obter a média das notas por turma e sexo, a consulta seria:

```sql
SELECT turma, sexo, AVG(nota) AS media FROM alunos GROUP BY turma, sexo;
```

Neste caso, cada combinação única de turma e sexo (por exemplo, "Turma A Masculino", "Turma A - Feminino", "Turma B - Masculino", etc.) se torna um grupo distinto, e a função `AVG(nota)` é aplicada a cada um desses subgrupos, fornecendo um resumo granular.

5.1.7. Função FLOOR()

A função `FLOOR()` no MySQL é uma função matemática que arredonda um número para baixo até o inteiro mais próximo. É particularmente útil quando você precisa categorizar dados numéricos em "faixas" inteiras.

Exemplos de uso:

- **SELECT FLOOR(8.75);** retornará 8
- **SELECT FLOOR(5.99);** retornará 5

- **SELECT FLOOR(3.00);** retornará 3

Essa função pode ser combinada com o `GROUP BY` para criar agrupamentos baseados em intervalos de valores, como veremos na prática.

5.2. Prática no MySQL Workbench

Agora que entendemos os conceitos teóricos, vamos colocar a mão na massa! Abriremos o MySQL Workbench e executaremos os comandos para ver como a agregação e o agrupamento funcionam na prática. Certifique-se de ter seu MySQL Workbench aberto e conectado ao seu banco de dados local.

5.2.1. Configuração Inicial

Abra o **MySQL Workbench.**

Clique na conexão **"Local MySQL"** (ou na sua conexão de banco de dados configurada).

Uma vez conectado, clique no campo **"Query 1"** (ou abra uma nova aba de consulta) para começar a inserir os comandos.

5.2.2. Funções Agregadas Básicas na Prática

Vamos começar com as funções agregadas básicas, relembrando o que cada comando faz e visualizando seus resultados.

1. **Média das Notas (AVG):**

- **Comando:** `SELECT AVG(nota) AS media_notas FROM alunos;`
- **Explicação:** Este comando calcula a média de todas as notas na tabela alunos e exibe o resultado com o apelido `media_notas`.
- **Ação:** Execute o comando (clicando no símbolo de raio) e observe a média das notas dos alunos.

1. **Maior Nota (MAX):**

- **Comando:** `SELECT MAX(nota) AS maior_nota FROM alunos;`
- **Explicação:** Encontra a maior nota na tabela alunos. A cláusula "AS" é usada para dar um apelido à coluna resultante, tornando-a mais legível. Por exemplo, se você alterar o apelido para "`MaiorNota`" e executar novamente, verá a mudança no cabeçalho da coluna.
- **Ação:** Execute o comando e observe a maior nota. Experimente mudar o apelido e reexecutar.

1. **Menor Nota (MIN):**

- **Comando:** `SELECT MIN(nota) AS menor_nota FROM alunos;`
- **Explicação:** Retorna a menor nota registrada na tabela apelido `menor_nota`.

- **Ação:** Execute o comando e verifique a menor nota.

1. **Soma das Notas (SUM):**

- **Comando:** `SELECT SUM(nota) AS soma_notas FROM alunos;`

- **Explicação:** Calcula a soma total de todas as notas na tabela alunos.

- **Ação:** Execute o comando e visualize a soma das notas.

1. **Contagem Total de Alunos (COUNT):**

- **Comando:** `SELECT COUNT(*) AS total_alunos FROM alunos;`

- **Explicação:** Conta o número total de registros (alunos) na tabela alunos.

- **Ação:** Execute o comando e observe o total de alunos.

5.2.3. Agrupamento na Prática

Vamos agora aplicar o `GROUP BY` para obter resumos por categoria.

1. **Alunos Ativos e Inativos:**

- **Comando:** `SELECT ativo, COUNT(*) AS total FROM alunos GROUP BY ativo;`

- **Explicação**: Este comando agrupa os alunos pelo status ativo (onde 1 pode representar ativo e 0 inativo) e conta quantos alunos estão em cada categoria.
- **Ação:** Execute o comando e observe a quantidade de alunos ativos e inativos.

5.2.4. WHERE vs. HAVING na Prática

Entender a diferença entre `WHERE` e `HAVING` é fundamental. Vamos ver como eles funcionam na prática.

1. **Filtragem Antes da Agregação (WHERE):**

- **Comando:** `SELECT AVG(nota) AS media_acima_de_5 FROM alunos WHERE nota > 5.0;`

- **Explicação:** Antes de calcular a média, este comando filtra os alunos, considerando apenas aqueles com nota superior a 5.0. A média é então calculada apenas para esses alunos selecionados.

- **Ação:** Execute o comando e veja a média das notas dos alunos que obtiveram mais de 5.0.

1. **Filtragem Depois da Agregação (HAVING):**

- **Comando:** `SELECT ativo, COUNT(*) AS total FROM alunos GROUP BY ativo HAVING COUNT(*) > 3;`

- **Explicação:** Primeiro, os alunos são agrupados pelo status ativo e a contagem de alunos é feita para cada grupo. Em seguida, o `HAVING` filtra esses grupos, mostrando apenas aqueles que possuem mais de 3 alunos.

- **Ação:** Execute o comando e observe os grupos que atendem à condição de ter mais de 3 alunos.

5.2.5. NULL e COUNT na Prática

Vamos explorar o comportamento de `COUNT` com valores `NULL`.

1. **Contando Notas Preenchidas (COUNT(coluna)):**

- **Comando:** `SELECT COUNT(nota) AS notas_preenchidas FROM alunos;`

- **Explicação:** Este comando conta apenas os registros onde a coluna não é `NULL`, ou seja, onde a nota foi preenchida.

- **Ação:** Execute o comando e observe o número de notas preenchidas.

1. **Contando Todos os Registros (COUNT(*)):**

- **Comando:** SELECT COUNT(*) AS total_registros FROM alunos;
- **Explicação:** Este comando conta todas as linhas na tabela incluindo aquelas onde a coluna nota pode ser alunos, `NULL`.
- **Ação:** Execute o comando e compare o resultado com o anterior. Você notará que `COUNT(*)` inclui registros com `NULL`.

5.2.6. Agrupamento por Múltiplas Colunas e FLOOR na Prática

Para finalizar a prática, vamos combinar o agrupamento por múltiplas colunas com a função `FLOOR()`.

1. **Agrupamento por Ativo e Faixa de Nota (com FLOOR):**

- **Comando:** `SELECT ativo, FLOOR(nota) AS nota_inteira, COUNT(*) AS total FROM alunos GROUP BY ativo, FLOOR(nota);`
- **Explicação:** Este comando agrupa os alunos primeiro pelo status em seguida, pela parte inteira da coluna ativo e, nota (arredondada para baixo com `FLOOR()`). Ele conta quantos alunos se encaixam em cada combinação de status e faixa de nota. Note que comandos longos podem ser divididos em várias linhas no MySQL, o que é uma prática comum para melhorar a legibilidade.

- **Ação:** Execute o comando e analise como os alunos são categorizados por status e faixas de notas inteiras.

5.3. Atividade prática da apostila

Vamos para nossa atividade prática da apostila para você praticar todos os conteúdos que aprendeu. Comece criando um banco de dados chamado `apostila5` e uma tabela chamada `atividade`, contendo os seguintes campos:

- `id` → inteiro, chave primária e auto incremento

- `nome` → texto (varchar 100)

- `nota` → decimal (5,2)

- `ativo` → inteiro (1 para ativo, 0 para inativo)

- `criado_em` → data/hora (timestamp)

Depois de criar o banco de dados e sua tabela, Insira pelo menos 20 registros na tabela atividade, variando notas acima e abaixo de 5.0, suas notas também devem variar com números inteiros e decimais, tendo alunos ativos e inativos, usando apelidos para ajudar. Agora que a tabela já está criada e com registros, siga as atividades:

1. Mostre a média das notas de todos os alunos.
2. Liste a maior nota registrada.
3. Conte quantos alunos existem no total.
4. Mostre quantos alunos estão ativos e quantos estão inativos.
5. Liste apenas os grupos de alunos (ativos ou inativos) que tenham mais de 3 alunos.
6. Mostre a média das notas somente dos alunos com nota maior que 5.0.
7. Conte quantos registros possuem nota preenchida (não nula).
8. Agrupe os alunos por nota inteira (sem casas decimais) e mostre quantos há em cada grupo.
9. Agrupe os alunos por status ativo e por nota inteira, mostrando quantos há em cada categoria.
10. Mostre a menor nota entre os alunos ativos.

Anotações

Olá, aluno(a)! Seja bem-vindo(a) à nossa sexta aula de banco de dados com MySQL. Nesta aula, aprofundaremos nosso conhecimento sobre JOINS e seus diversos comandos.

6.1. O que são JOINS?

Para entender o que são JOINS, imagine a seguinte situação: você tem duas listas de informações em papel. Uma lista contém dados de alunos (como nome e ID), e a outra lista contém dados de matrículas (como o ID do aluno e o curso matriculado). Sozinhas, cada lista fornece informações limitadas. No entanto, se você deseja saber qual aluno está matriculado em qual curso, você precisa combinar essas duas listas usando um dado em comum: o ID do aluno.

Essa operação de "juntar informações de duas (ou mais) tabelas com base em um campo em comum" é exatamente o que chamamos de JOIN no SQL. O MySQL oferece diferentes tipos de JOINs, permitindo que você combine essas listas de maneiras distintas, dependendo do resultado que deseja obter.

6.1.1. Cenário de Exemplo: Alunos e Matrículas

Para ilustrar os conceitos de JOINs, utilizaremos duas tabelas hipotéticas:

- ***alunos:*** Contém informações sobre os estudantes. **Exemplo:** Ana (ID 1), Carlos (ID 2), Beatriz (ID 3)
- ***matriculas:*** Contém informações sobre os cursos em que os alunos estão matriculados. **Exemplo:** Matemática (matricula_id 1, aluno_id 1 - Ana), História (matricula_id 2, aluno_id 3 - Beatriz)

Com este cenário em mente, exploraremos os tipos de JOINs e como eles nos ajudam a extrair informações valiosas combinando dados dessas tabelas.

6.2. A Importância da Cláusula ON nos JOINS

Antes de mergulharmos nos tipos específicos de JOINs, é crucial entender a cláusula ***ON***. Quando você utiliza um ***JOIN*** para unir duas tabelas, é necessário especificar como elas devem se relacionar, ou seja, qual coluna de uma tabela corresponde a qual coluna da outra tabela. Essa correspondência é definida pela cláusula ***ON***.

Sem a cláusula ***ON***, o banco de dados não saberia quais linhas devem ser combinadas, resultando em um comportamento imprevisível ou em um produto cartesiano, onde cada linha da primeira tabela é combinada com cada linha da segunda tabela, gerando um resultado gigante e, na maioria das vezes, sem sentido.

6.2.1. Exemplo Prático da Cláusula ON

Considerando nossas tabelas ***alunos*** (com ***id*** do aluno) e ***matriculas*** (com ***aluno_id*** referenciando o ***id*** do aluno), se quisermos juntar essas tabelas para ver o nome do aluno junto com o curso em que ele está matriculado, precisamos dizer ao SQL:

```sql
ON alunos.id = matriculas.aluno_id
```

Essa expressão indica que as linhas devem ser combinadas quando o ***id*** da tabela ***alunos*** for igual ao ***aluno_id*** da tabela ***matriculas***. A cláusula ***ON*** mostra "quais linhas da Tabela A têm correspondência com quais linhas da Tabela B".

6.2.2. Por que a Cláusula ON é tão importante?

A cláusula *ON* é fundamental por três motivos principais:

1. **Juntar as informações corretas:** Sem *ON*, o banco de dados poderia combinar qualquer aluno com qualquer matrícula, o que não faria sentido para a integridade dos dados.

1. **Evitar resultados estranhos (Produto Cartesiano):** Como mencionado, a ausência ou uso incorreto do *ON* pode levar a um produto cartesiano, onde todas as linhas de uma tabela são combinadas com todas as linhas da outra, gerando um volume de dados desnecessário e incorreto.

1. **Ler o resultado certo:** A cláusula *ON* permite identificar com clareza as relações entre as tabelas, possibilitando consultas precisas, como, por exemplo, quais alunos têm matrícula ou quais ainda não têm.

Compreendida a importância da cláusula *ON*, estamos prontos para explorar os principais tipos de JOINs.

6.3. Tipos de JOINS no MySQL

O MySQL oferece diferentes tipos de JOINs, cada um com uma finalidade específica para combinar dados de tabelas. Vamos detalhar os mais comuns:

6.3.1. INNER JOIN (ou apenas JOIN)

O *INNER JOIN* (ou simplesmente *JOIN*) retorna apenas os registros que possuem correspondência em ambas as tabelas. Ele funciona como uma interseção entre os conjuntos de dados das duas tabelas.

História para entender: Imagine duas listas de nomes: uma de pessoas que gostam de café e outra de pessoas que gostam de chá. O INNER JOIN mostraria apenas as pessoas que aparecem nas duas listas, ou seja, aquelas que gostam tanto de café quanto de chá. Quem está só em uma lista, fica de fora.

Exemplo com *alunos* e *matriculas*:

sql

```sql
SELECT a.id, a.nome, m.curso
FROM alunos a
INNER JOIN matriculas m ON a.id =
m.aluno_id;
```

Explicação:

- ***a*** é o apelido para a tabela ***alunos*** e ***m*** para a tabela ***matriculas***.
- **A cláusula *ON a.id = m.aluno_id* define a condição de junção:** o id do aluno na tabela alunos deve ser igual ao aluno_id na tabela matriculas.
- **Resultado:** Este comando retornará apenas os alunos que possuem uma matrícula registrada. Se um aluno não tiver matrícula, ele não aparecerá no resultado. Da mesma forma, se houver uma matrícula cujo ***aluno_id*** não existe na tabela ***alunos***, essa matrícula também não será exibida, pois o *INNER JOIN* exige correspondência em ambas as tabelas.

6.3.2. LEFT JOIN (ou LEFT OUTER JOIN)

O *LEFT JOIN* (ou *LEFT OUTER JOIN*) retorna **todas as linhas da "tabela da esquerda"** (a que vem antes do *LEFT JOIN*) e os dados correspondentes da tabela da direita. Se não houver correspondência na tabela da direita, os campos dessa tabela virão como *NULL*.

História para entender: Imagine que você tem uma lista completa de todos os alunos da escola. Você quer saber em qual curso cada aluno está matriculado. O *LEFT JOIN* pegaria a lista de todos os alunos e, para cada um, tentaria encontrar uma matrícula. Se encontrar, mostra o curso; se não encontrar, o aluno ainda aparece na lista, mas a coluna do curso fica em branco (NULL).

Exemplo com *alunos* e *matriculas*:

sql

```sql
SELECT a.id, a.nome, m.curso
FROM alunos a
LEFT JOIN matriculas m ON a.id = m.aluno_id;
```

Explicação:

- Este comando mostrará todos os alunos, mesmo aqueles que não possuem matrícula. Para os alunos sem matrícula, as colunas vindas da tabela ***matriculas*** (como ***curso***) aparecerão como ***NULL***.
- É útil quando você deseja ver todos os registros de uma tabela (a esquerda) e, se houver, as informações correspondentes da outra tabela.

6.3.3. RIGHT JOIN (ou RIGHT OUTER JOIN)

O ***RIGHT JOIN*** (ou ***RIGHT OUTER JOIN***) é o oposto do ***LEFT JOIN***. Ele retorna **todas as linhas da "tabela da direita"** (a que vem depois do ***RIGHT JOIN***) e as correspondências da tabela da esquerda. Se não houver correspondência na tabela da esquerda, os campos dessa tabela virão como ***NULL***.

História para entender: Imagine que você tem uma lista completa de todas as matrículas feitas na escola. Você quer saber qual aluno fez cada matrícula. O ***RIGHT JOIN*** pegaria a lista de todas as matrículas e, para cada uma, tentaria encontrar o aluno correspondente. Se encontrar, mostra o nome do aluno; se não encontrar (por exemplo, se a matrícula se refere a um ***aluno_id*** que não existe mais), a coluna do nome do aluno fica em branco (NULL).

Exemplo com alunos e matriculas:

sql

```sql
SELECT a.id, a.nome, m.curso
FROM alunos a
RIGHT JOIN matriculas m ON a.id =
m.aluno_id;
```

Explicação:

- Este comando mostrará todas as matrículas, mesmo que o aluno correspondente na tabela ***alunos*** não exista. Se uma matrícula apontar para um ***aluno_id*** que não está em ***alunos***, os campos de ***alunos*** (como ***nome***) aparecerão como ***NULL*** para essa linha.
- Matrículas que tiverem um aluno correspondente mostrarão o nome e os dados do aluno normalmente.
- Embora o MySQL permita ***RIGHT JOIN***, muitas vezes é preferível reorganizar as tabelas e usar ***LEFT JOIN*** para manter a clareza e a consistência na leitura do código, pois a maioria dos desenvolvedores tende a pensar da esquerda para a direita.

6.3.4. CROSS JOIN (Produto Cartesiano)

O ***CROSS JOIN*** retorna todas as combinações possíveis entre as linhas da Tabela A e as linhas da Tabela B. Cada linha da Tabela A é combinada com cada linha da Tabela B.

História para entender: Imagine que você tem uma lista de cores (vermelho, azul) e uma lista de tamanhos (P, M, G). Se você fizer um ***CROSS JOIN***, o resultado será todas as combinações possíveis: (vermelho, P), (vermelho, M), (vermelho, G), (azul, P), (azul, M), (azul, G).

Exemplo com alunos e matriculas:

sql

```sql
SELECT a.nome, m.curso
FROM alunos a
CROSS JOIN matriculas m;
```

Explicação:

- Se houver 5 alunos e 3 matrículas, o resultado terá ***5 × 3 = 15*** linhas, mostrando todas as combinações possíveis de nome de aluno com curso.
- O ***CROSS JOIN*** é raramente útil para consultas de dados relacionadas, mas pode ser empregado para gerar todas as

possibilidades ou para fins de teste. Use-o apenas quando realmente precisar combinar tudo com tudo, sem uma condição de correspondência específica.

6.3.5. SELF JOIN (Auto-Join)

O *SELF JOIN* é uma técnica onde uma tabela é unida a **ela mesma**. Isso é útil quando você precisa comparar registros dentro da própria tabela, como encontrar amigos em uma lista de pessoas, identificar supervisores em uma hierarquia de funcionários, ou analisar relacionamentos internos.

História para entender: Suponha que na sua lista de alunos, alguns alunos são tutores de outros alunos. Ambos são alunos, mas você quer ver a relação de tutoria. Você "junta" a tabela ***alunos*** com ela mesma, mas usando apelidos diferentes para cada "cópia" da tabela, permitindo a comparação.

Exemplo com *alunos* (supondo um campo *mentor_id*):

Suponha que a tabela alunos tenha um campo ***mentor_id*** que referencia o ***id*** de outro aluno dentro da mesma tabela.

sql

```sql
SELECT a.nome AS aluno, b.nome AS mentor
FROM alunos a
LEFT JOIN alunos b ON a.mentor_id = b.id;
```

Explicação:

- Neste comando, ***a*** e ***b*** são, na verdade, a mesma tabela ***alunos***, mas atuam como se fossem duas tabelas distintas para permitir a comparação.
- O ***LEFT JOIN*** é usado para garantir que todos os alunos sejam listados, mesmo aqueles que não possuem um ***mentor_id*** ou cujo ***mentor_id*** não corresponde a um id existente.
- **Resultado:** Você veria uma lista de alunos e, ao lado, o nome de seu mentor (se houver).

Outro Exemplo (do roteiro, para matrículas):

Para ver a progressão de cursos de um mesmo aluno, comparando matrículas atuais com matrículas anteriores:

sql

```sql
SELECT m1.aluno_id, m1.curso AS curso_atual,
m2.curso AS curso_anterior
FROM matriculas m1
LEFT JOIN matriculas m2 ON m1.aluno_id =
m2.aluno_id
AND m2.data_matricula < m1.data_matricula;
```

Explicação:

- Aqui, a tabela matriculas é unida a si mesma (***m1*** e ***m2***).
- A condição ***ON m1.aluno_id = m2.aluno_id*** garante que estamos comparando matrículas do mesmo aluno.
- A condição ***AND m2.data_matricula < m1.data_matricula*** filtra para que ***m2*** represente uma matrícula anterior a ***m1***.
- **Resultado:** Cada matrícula "atual" (***m1***) aparece com sua matrícula anterior (***m2***), se existir, para o mesmo aluno. Se um aluno tiver apenas uma matrícula, a coluna ***curso_anterior*** aparecerá como ***NULL***.

6.3.6. FULL OUTER JOIN (Simulação no MySQL)

Um ***FULL OUTER JOIN*** retornaria todas as linhas de ambas as tabelas, com correspondências quando existirem, e preenchendo com ***NULL*** quando não houver correspondência. Ele é, essencialmente, a união de um ***LEFT JOIN*** e um ***RIGHT JOIN***.

Importante: O MySQL **não possui** o ***FULL OUTER JOIN*** implementado diretamente como um comando único. Para simular o comportamento de um ***FULL OUTER JOIN*** no MySQL, geralmente utilizamos a combinação de ***LEFT JOIN*** e ***RIGHT JOIN*** com o operador ***UNION***.

Exemplo de Simulação:

sql

```sql
SELECT a.id, a.nome, m.curso
FROM alunos a
LEFT JOIN matriculas m ON a.id = m.aluno_id

UNION

SELECT a.id, a.nome, m.curso
FROM alunos a
RIGHT JOIN matriculas m ON a.id = m.aluno_id
WHERE a.id IS NULL; -- Para evitar
duplicatas de correspondências exatas
```

Explicação:

- A primeira parte (***LEFT JOIN***) traz todos os alunos e suas matrículas correspondentes, preenchendo com ***NULL*** para alunos sem matrícula.

- A segunda parte (***RIGHT JOIN*** com ***WHERE a.id IS NULL***) traz as matrículas que não têm um aluno correspondente, preenchendo com ***NULL*** para os dados do aluno. O ***WHERE a.id IS NULL*** é crucial para evitar duplicar as linhas que já foram retornadas pelo ***LEFT JOIN*** (onde há correspondência).

- O operador ***UNION*** combina os resultados de ambas as consultas, removendo automaticamente as linhas duplicadas.

Não se preocupe se o conceito de ***UNION*** ainda não é familiar; ele será abordado em detalhes em uma aula futura.

6.4. Desafio Prático: Criando e Manipulando Tabelas com JOINS

Para consolidar o aprendizado sobre JOINS, vamos realizar um desafio prático no MySQL Workbench. Iremos criar uma nova tabela e, em seguida, aplicar todos os comandos de JOIN que estudamos.

6.4.1. Preparando o Ambiente: MySQL Workbench

1. Abra o **MySQL Workbench**.
2. Conecte-se ao seu banco de dados local (ex: **local mysql**).
3. Certifique-se de que a aba **Query 1** esteja aberta para digitar os comandos.

6.4.2. Criando a Tabela matriculas

Para nossos exemplos, precisamos de uma tabela ***matriculas*** que se relacione com a tabela ***alunos*** (que já deve existir em seu banco de dados). Vamos criá-la com as seguintes colunas:

- ***matricula_id:*** Identificador único da matrícula (chave primária, auto-incremento).

- ***aluno_id:*** Referência ao ***id*** da tabela ***alunos*** (chave estrangeira).

- ***curso***: Nome do curso/matéria.
- ***data_matricula:*** Data e hora em que a matrícula foi feita (com valor padrão para o momento atual).
- ***ativa:*** Indica se a matrícula está ativa (booleano, com valor padrão ***TRUE***).

Comandos para criar a tabela *matriculas*:

sql

```sql
CREATE TABLE matriculas (
    matricula_id INT AUTO_INCREMENT PRIMARY
KEY,
    aluno_id INT NOT NULL,
    curso VARCHAR(100) NOT NULL,
    data_matricula TIMESTAMP DEFAULT
CURRENT_TIMESTAMP,
    ativa BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (aluno_id) REFERENCES
alunos(id)
);
```

Explicação da FOREIGN KEY:

O comando ***FOREIGN KEY (aluno_id) REFERENCES alunos(id)*** é crucial. Ele define uma chave estrangeira na tabela ***matriculas***. Isso significa que a coluna ***aluno_id*** na tabela ***matriculas*** deve obrigatoriamente conter um valor que já exista na coluna ***id*** da tabela ***alunos***.

Por que a Chave Estrangeira é importante?

Ela garante a ***integridade referencial*** do seu

banco de dados, evitando que você crie matrículas para alunos que não existem. Veja alguns exemplos de como ela atua:

- **Inserção:** Se você tentar inserir uma matrícula com ***aluno_id = 99***, mas não houver um aluno com ***id = 99*** na tabela ***alunos***, o MySQL gerará um erro e impedirá a operação.
- **Exclusão:** Se você tentar deletar um aluno que ainda possui matrículas associadas, o banco de dados pode recusar a operação (dependendo da configuração **ON DELETE**), prevenindo a criação de registros "órfãos" (matrículas sem um aluno válido).
- **Atualização:** Se o ***id*** de um aluno for atualizado para um valor que desconectaria suas matrículas, o banco pode impedir a atualização ou propagar a mudança (dependendo das opções ***ON UPDATE***).

O ***ON DELETE*** é uma parte opcional da declaração de chave estrangeira que especifica o que o banco de dados deve fazer automaticamente quando uma linha da tabela "pai" (neste caso, ***alunos***) é deletada.

Execute o comando ***CREATE TABLE*** no MySQL Workbench para criar a tabela ***matriculas***.

6.4.3. Adicionando Registros à Tabela matriculas

Agora que a tabela ***matriculas*** foi criada, vamos adicionar alguns registros para que possamos testar os JOINS. Observe os comandos abaixo:

sql

```sql
INSERT INTO matriculas (aluno_id, curso,
ativa) VALUES
(3, 'Matemática', TRUE),
(3, 'Física', TRUE),
(5, 'História', TRUE),
(6, 'Matemática', FALSE),
(10, 'Química', TRUE),
(20, 'Biologia', TRUE);
```

Atenção ao Erro Intencional!

Ao executar este comando, você notará um erro. Isso é intencional para demonstrar a funcionalidade da ***FOREIGN KEY*** que acabamos de configurar. O erro ocorre porque estamos tentando inserir um registro de ***Biologia*** com ***aluno_id = 20***, mas não existe um aluno com ***id = 20*** na tabela ***alunos***. A ***FOREIGN KEY*** impede essa inserção, garantindo a integridade dos dados.

Correção do Erro:

Para corrigir, remova a linha ***(20, 'Biologia', TRUE);*** ou substitua o ***aluno_id*** por um ID válido que exista na sua tabela ***alunos***. Após a correção, execute o comando ***INSERT*** novamente.

6.4.4. Aplicando os Comandos JOINS na Prática

Com as tabelas alunos e matriculas populadas, vamos aplicar os diferentes tipos de JOINs.

6.4.4.1. INNER JOIN

Comando:

```sql
SELECT a.id AS aluno_id, a.nome, m.curso,
m.ativa AS matricula_ativa
FROM alunos a
INNER JOIN matriculas m ON a.id =
m.aluno_id;
```

Explicação: Este comando retornará apenas os alunos que possuem uma matrícula registrada. Alunos sem matrícula não aparecerão, pois o ***INNER JOIN*** exige correspondência em ambas as tabelas. Da mesma forma, matrículas sem um ***aluno_id*** correspondente na tabela ***alunos*** também não serão exibidas.

Execute o comando e observe o resultado.

6.4.4.2. LEFT JOIN

Comando:

sql

```sql
SELECT a.id AS aluno_id, a.nome, m.curso,
m.ativa AS matricula_ativa
FROM alunos a
LEFT JOIN matriculas m ON a.id = m.aluno_id;
```

Explicação: Todos os alunos serão mostrados, sem exceção. Se um aluno não tiver matrícula correspondente, os campos vindos da tabela ***matriculas*** (*curso*, ***matricula_ativa***) aparecerão como ***NULL***. Este comando é útil quando você quer ver todos os alunos, mesmo os que ainda não fizeram matrícula, e mostrar a matrícula quando ela existir.

Execute o comando e observe o resultado.

6.4.4.3. RIGHT JOIN

Comando:

sql

```sql
SELECT a.id AS aluno_id, a.nome, m.curso,
m.ativa AS matricula_ativa
FROM alunos a
RIGHT JOIN matriculas m ON a.id =
m.aluno_id;
```

Explicação: Todas as linhas da tabela ***matriculas*** aparecerão, mesmo que o aluno correspondente na tabela ***alunos*** não exista. Se uma matrícula apontar para um ***aluno_id*** que não existe em ***alunos***, os campos de ***alunos*** (como ***nome***) aparecerão como ***NULL***. Matrículas que tiverem um aluno correspondente mostrarão o nome e os dados do aluno normalmente.

Execute o comando e observe o resultado.

6.4.4.4. CROSS JOIN

Comando:

sql

```sql
SELECT a.nome, m.curso
FROM alunos a
CROSS JOIN matriculas m;
```

Explicação: Se você tiver, por exemplo, 10 alunos e 6 matrículas, o resultado será **10 × 6 = 60** linhas, mostrando todas as combinações possíveis de nome de aluno com curso. Este tipo de resultado raramente é útil sozinho, mas serve para entender como o combinatório funciona ou gerar todas as possibilidades.

Execute o comando e observe o resultado.

6.4.4.5. SELF JOIN

Comando (Exemplo para matrículas):

sql

```sql
SELECT m1.aluno_id, m1.curso AS curso_atual,
m2.curso AS curso_anterior
FROM matriculas m1
LEFT JOIN matriculas m2 ON m1.aluno_id =
m2.aluno_id
AND m2.data_matricula < m1.data_matricula;
```

Explicação: Este comando mostra, para cada matrícula de um aluno (***m1***), qual foi a matrícula anterior dele (***m2***), baseando-se na data. Isso ajuda a ver a progressão de cursos de um mesmo aluno. Se um aluno tiver apenas uma matrícula, a parte de ***curso_anterior*** aparecerá como ***NULL***. O ***SELF JOIN*** é poderoso para comparar registros dentro da mesma tabela, comparar histórico, ou relacionar pares.

Execute o comando e observe o resultado.

6.5. Atividades Práticas no MySQL Workbench

Com base no nosso estudo aprofundado sobre **JOINs** e seus comandos (INNER, LEFT, RIGHT, CROSS e SELF), vamos agora aplicar esse conhecimento em um **novo cenário de banco de dados** que simula uma estrutura empresarial.

Você deverá criar um novo banco de dados e duas tabelas que sejam compatíveis entre si, de modo que possamos utilizar todos os tipos de JOINs que aprendemos.

6.5.1. Criação do Banco de Dados

Primeiro, crie um novo banco de dados para

esta aula chamado ***apostila6***.

6.5.2. Criação das Tabelas

Crie as tabelas ***funcionarios*** e ***trabalhos*** com as seguintes estruturas, garantindo que a coluna ***func_id*** na tabela ***trabalhos*** seja uma **chave estrangeira** que referencia a tabela ***funcionarios:***

1. **funcionarios:** Armazenará informações básicas dos colaboradores.

1. **trabalhos:** Armazenará os projetos ou tarefas aos quais os funcionários estão designados (simulando a relação de 1 para N: um funcionário pode ter vários trabalhos).

Tabela: funcionários

COLUNA | TIPO | RESTRIÇÕES | DESCRIÇÃO

Id INT PRIMARY KEY AUTO_INCREMENT - Identificador único do funcionário

nome VARCHAR(100) NOT NULL - Nome completo do funcionário

cargo VARCHAR(50) NOT NULL - Cargo/Função Atual

supervisor_id INT (opcional) ID do supervisor (para SELF JOIN)

Tabela: trabalhos

COLUNA | TIPO | RESTRIÇÕES | DESCRIÇÃO

cod_trabalho INT PRIMARY KEY AUTO_INCREMENT - Código único do trabalho/projeto

func_id INT NOT NULL FOREIGN KEY (referencia funcionarios.id) - ID do funcionário responsável.

projeto VARCHAR(100) NOT NULL - Nome ou descrição do projeto.

data_entrega DATE NOT NULL - Data prevista de entrega.

6.5.3. Inserção de Dados

Para que possamos testar todos os JOINs (incluindo aqueles que retornam NULLs), insira um conjunto de dados que atenda aos seguintes critérios na tabela funcionarios e trabalhos:

- Insira **pelo menos 6 funcionários.** Certifique-se de que alguns funcionários tenham um ***supervisor_id*** referenciando outro funcionário (para o SELF JOIN).
- Insira **pelo menos 6 trabalhos.**
- Garanta que **alguns funcionários tenham 2 ou mais trabalhos.**
- Garanta que **pelo menos um funcionário não tenha nenhum trabalho** associado (para o LEFT JOIN).
- Garanta que **pelo menos um trabalho seja associado a um *func_id* que não existe** na tabela ***funcionarios*** (se o MySQL permitir, para o RIGHT JOIN, embora na prática o FOREIGN KEY deve impedir isso, tente um ID alto que não existe para ver o erro e depois insira um válido para os testes).

6.5.4. Atividades Práticas com JOINs

Agora que suas tabelas estão criadas e populadas, realize as seguintes consultas e me envie os comandos SQL (e, se possível, os resultados):

6.5.4.1. Atividade: INNER JOIN

Objetivo: Exibir apenas os funcionários que estão atualmente designados a algum trabalho/projeto.

Ação: Escreva uma consulta que liste o nome do funcionário, o cargo e o projeto em que ele está trabalhando, mas apenas se houver correspondência nas duas tabelas.

6.5.4.2. Atividade: LEFT JOIN

Objetivo: Exibir todos os funcionários, listando seus projetos se existirem.

Ação: Liste o nome do funcionário e o projeto. Funcionários sem projeto devem ter o campo projeto como NULL (ou em branco).

6.5.4.3. Atividade: RIGHT JOIN

Objetivo: Listar todos os projetos, exibindo o nome do funcionário responsável.

Ação: Liste o projeto, a data de entrega e o nome do funcionário. Se houver projetos sem um func_id válido (ou se você quiser garantir que todos os projetos apareçam), os dados do funcionário devem vir como NULL.

6.5.4.4. Atividade: CROSS JOIN

Objetivo: Gerar todas as combinações possíveis de funcionários e projetos.

Ação: Escreva uma consulta que combine cada nome de funcionário com cada projeto existente na tabela trabalhos.

6.5.4.5. Atividade: SELF JOIN

Objetivo: Identificar quem é o supervisor de cada funcionário.

Ação: Escreva uma consulta que liste o nome do funcionário e o nome de seu supervisor (o qual também é um funcionário na mesma tabela).

Anotações

Olá! Seja bem-vindo(a) à nossa aula dedicada ao aprofundamento em Subconsultas e Operadores de Conjunto no MySQL. Nesta apostila, exploraremos esses conceitos fundamentais que permitem realizar consultas mais complexas e eficientes em seus bancos de dados. Entenderemos como formular perguntas aninhadas e como combinar resultados de diferentes consultas para obter informações valiosas.

7.1. Subconsultas (Subqueries) no MySQL

7.1.1. O que são Subconsultas?

Uma **subconsulta** (ou subquery) é, em sua essência, uma consulta SQL aninhada dentro de outra consulta SQL. Pense nela como uma **pergunta secundária** que você faz ao banco de dados para ajudar a responder a uma **pergunta principal**. Essa capacidade de aninhar consultas torna o SQL extremamente poderoso, permitindo resolver problemas complexos de forma elegante.

As subconsultas podem retornar:

- Uma **lista de valores** (por exemplo, IDs de alunos).
- Um **único valor** (por exemplo, a média de notas).
- Uma **pequena “tabela temporária”**.

Elas podem ser utilizadas em diversas cláusulas de uma consulta principal, como ***WHERE***, **SELECT**, **FROM** e até mesmo ***HAVING***.

7.1.2. Tipos Comuns de Subconsultas

Vamos explorar os tipos mais comuns de subconsultas com exemplos práticos, utilizando tabelas hipotéticas ***alunos*** (com ***id***, ***nome***, ***nota***, ***ativo***) e ***matriculas*** (com ***aluno_id***, ***curso***, ***ativa***).

7.1.2.1. Subconsulta no WHERE com IN

Este tipo de subconsulta é usado para comparar uma coluna com uma lista de valores retornada pela subconsulta. É extremamente útil para **filtrar registros com base em um conjunto de resultados.**

Exemplo: Encontrar os nomes dos alunos matriculados no curso de 'Matemática'.

sql

```sql
SELECT nome
FROM alunos
WHERE id IN (
    SELECT aluno_id
    FROM matriculas
    WHERE curso = 'Matemática'
);
```

Explicação:

1. A subconsulta ***(SELECT aluno_id FROM matriculas WHERE curso = 'Matemática')*** primeiro identifica todos os ***aluno_id*** que estão matriculados no curso de 'Matemática'.
2. A consulta externa ***SELECT nome FROM alunos WHERE id IN (...)*** então utiliza essa lista de ***aluno_id*** para buscar os nomes correspondentes na tabela ***alunos***.

É como dizer: “Me diga os nomes dos alunos que estão na lista de quem faz Matemática”.

7.1.2.2. Subconsulta com EXISTS

O operador ***EXISTS*** verifica a existência de linhas que satisfaçam uma condição dentro da subconsulta. Ele retorna ***TRUE*** se a subconsulta retornar pelo menos uma linha, e ***FALSE*** caso contrário. É geralmente mais eficiente que ***IN***

quando a única preocupação é a existência de dados.

Exemplo: Listar os nomes dos alunos que possuem pelo menos uma matrícula no curso de 'Física'.

sql

```sql
SELECT nome
FROM alunos a
WHERE EXISTS (
    SELECT 1
    FROM matriculas m
    WHERE m.aluno_id = a.id
    AND m.curso = 'Física'
);
```

Explicação:

1. Para cada aluno na tabela alunos (referenciado como ***a***), a subconsulta verifica se existe alguma matrícula (***m***) para aquele aluno (***m.aluno_id = a.id***) no curso de 'Física'.

1. Se a subconsulta encontrar uma ou mais matrículas, ***EXISTS*** retorna ***TRUE***, e o nome do aluno é incluído no resultado.

Note o uso de ***a*** como alias para a tabela ***alunos*** na consulta externa, e ***m*** para ***matriculas*** na subconsulta. Este é um exemplo de **subconsulta correlacionada**, onde a subconsulta referencia colunas da consulta externa.

7.1.2.3. Subconsulta Escalar

Uma **subconsulta escalar** é aquela que retorna um único valor (uma única célula). Ela pode ser utilizada tanto na cláusula ***SELECT*** quanto na cláusula ***WHERE***.

Exemplo: Mostrar o nome de cada aluno e a quantidade total de matrículas que ele possui.

sql

```sql
SELECT a.nome,
       (SELECT COUNT(*)
        FROM matriculas m
        WHERE m.aluno_id = a.id) AS
qtd_matriculas
FROM alunos a;
```

Explicação:

1. Para cada linha da tabela alunos (referenciada como **a**), a subconsulta ***(SELECT COUNT(*) FROM matriculas m WHERE m.aluno_id = a.id)*** é executada.
2. Essa subconsulta conta o número de matrículas (***COUNT(*)***) associadas ao ***id*** do aluno atual (***a.id***).
3. O resultado, um único número, é então exibido na coluna ***qtd_matriculas*** ao lado do nome do aluno.

7.1.2.4. Subconsulta no FROM (Tabela Derivada)

Quando uma subconsulta é utilizada na cláusula ***FROM***, ela se comporta como uma **tabela temporária** para a consulta externa. O resultado da subconsulta é tratado como uma nova tabela, da qual a consulta principal pode selecionar dados.

Exemplo: Listar os IDs dos alunos que possuem mais de uma matrícula.

sql

```sql
SELECT t.aluno_id, t.qtd
FROM (
    SELECT aluno_id, COUNT(*) AS qtd
    FROM matriculas
    GROUP BY aluno_id
) AS t
WHERE t.qtd > 1;
```

Explicação:

1. A subconsulta ***(SELECT aluno_id, COUNT(*) AS qtd FROM matriculas GROUP BY aluno_id)*** agrupa as matrículas por ***aluno_id*** e conta a quantidade de matrículas para

cada aluno, criando uma tabela temporária com ***aluno_id*** e ***qtd***.
2. Essa tabela temporária é apelidada de ***t***.
3. A consulta externa ***SELECT t.aluno_id, t.qtd FROM (...) AS t WHERE t.qtd > 1*** então seleciona os ***aluno_id*** e ***qtd*** dessa tabela ***t*** onde a quantidade de matrículas (***qtd***) é maior que 1.

7.1.3. Subconsultas Correlacionadas vs. Não Correlacionadas

É importante distinguir entre esses dois tipos de subconsultas, pois eles têm implicações significativas no desempenho.

- **Subconsulta Não Correlacionada:** É independente da consulta externa. Ela pode ser executada por si só e retorna um resultado que é então usado pela consulta externa. Exemplo: ***SELECT aluno_id FROM matriculas WHERE curso = 'Matemática'***. O resultado é fixo e a subconsulta é executada apenas uma vez.

- **Subconsulta Correlacionada:** Referencia colunas da consulta externa e, portanto, precisa ser reavaliada para cada linha da consulta externa. Isso significa que a subconsulta é executada múltiplas vezes, uma para cada linha processada pela consulta principal.

Exemplo de Subconsulta Correlacionada: Encontrar alunos cuja nota é superior à média de notas dos alunos com o mesmo status (ativo ou inativo).

sql
```sql
SELECT a.nome, a.nota
FROM alunos a
WHERE a.nota > (
    SELECT AVG(nota)
    FROM alunos
    WHERE ativo = a.ativo
);
```

Explicação: A subconsulta ***(SELECT AVG(nota) FROM alunos WHERE ativo = a.ativo)*** usa ***a.ativo***, que é uma coluna da consulta externa. Para cada aluno ***a***, a subconsulta calcula a média das notas dos alunos com o mesmo status ***ativo*** que ***a***. Por depender da linha atual de ***a***, ela é executada repetidamente. Subconsultas correlacionadas podem ser mais lentas em tabelas grandes.

7.1.4. Vantagens e Cuidados com Subconsultas

Vantagens:

- **Facilidade de Leitura:** Quando a lógica é naturalmente aninhada, as subconsultas podem tornar o código mais legível e compreensível.
- **Resolução de Problemas Específicos:** São úteis para perguntas do tipo "dentro desse conjunto, escolha X".

Cuidados:

- **Desempenho:** Subconsultas correlacionadas, em particular, podem ser lentas em grandes bases de dados. Muitas vezes, é preferível reescrevê-las usando ***JOINs*** ou tabelas derivadas para otimizar o desempenho.
- **Restrições em *UPDATE*:** Em alguns casos no MySQL, há restrições para atualizar e selecionar a mesma tabela em subqueries dentro de um ***UPDATE***. Se ocorrer um erro, a regra prática é reescrever a operação usando um ***JOIN*** ou uma tabela temporária.

7.2. Operadores de Conjunto no MySQL

Os **operadores de conjunto** permitem combinar os resultados de duas ou mais consultas ***SELECT*** em um único conjunto de resultados. Eles tratam cada ***SELECT*** como uma "lista" e unem essas listas.

7.2.1. UNION e UNION ALL

São os operadores de conjunto mais comuns.

- ***UNION:*** Combina os resultados de duas ou mais consultas ***SELECT*** e remove automaticamente as linhas duplicadas.

- ***UNION ALL:*** Combina os resultados de duas ou mais consultas ***SELECT*** e **mantém todas as linhas, incluindo as duplicadas.** É geralmente mais rápido que ***UNION*** porque não precisa realizar a etapa de remoção de duplicatas.

Exemplo: Criar uma lista única de IDs de alunos que têm nota maior ou igual a 9 OU que estão matriculados no curso de 'Matemática'.

```sql
SELECT id FROM alunos WHERE nota >= 9
UNION
SELECT aluno_id FROM matriculas WHERE curso
= 'Matemática';
```

Explicação:

1. A primeira consulta seleciona os ***id***s dos alunos com nota >= 9.
2. A segunda consulta seleciona os ***aluno_id***s dos alunos matriculados em 'Matemática'.
3. O ***UNION*** combina essas duas listas, e se um ***id*** aparecer em ambas, ele será listado apenas uma vez no resultado final.

Se o mesmo ***id*** aparecer nas duas consultas, com ***UNION*** ele aparecerá apenas uma vez; com ***UNION ALL*** aparecerá duas vezes.

7.2.2. Regras Importantes para UNION/UNION ALL

1. **Número e Tipos de Colunas:** As consultas combinadas devem ter o **mesmo número de colunas** e os **tipos de dados das colunas correspondentes devem ser compatíveis** (ou convertíveis). Por exemplo, ***SELECT nome FROM alunos UNION SELECT curso FROM matriculas*** funciona se ambas as colunas (***nome*** e ***curso***) forem do tipo texto.
2. ***ORDER BY:*** A cláusula ***ORDER BY*** só pode ser usada **após o último *SELECT*** para ordenar o conjunto combinado de resultados.
3. **Custo de Desempenho:** ***UNION*** tem um custo de desempenho maior devido à remoção de duplicatas. Use ***UNION ALL*** se você não precisar dessa remoção, pois será mais rápido.

7.2.3. Outros Operadores de Conjunto (INTERSECT e EXCEPT/MINUS)

Embora ***UNION*** e ***UNION ALL*** sejam os mais comuns, outros operadores de conjunto existem em alguns sistemas de gerenciamento de banco de dados (SGBDs):

- ***INTERSECT:*** Retorna apenas as linhas que aparecem em **ambas** as consultas.
- ***EXCEPT*** (ou ***MINUS*** em alguns SGBDs como Oracle): Retorna as linhas da **primeira consulta que não estão presentes na segunda consulta.**

Historicamente, nem todas as versões do MySQL ofereceram suporte direto a ***INTERSECT*** e ***EXCEPT***. Em situações onde esses operadores não estão disponíveis, é comum recorrer a ***JOIN***s (como ***INNER JOIN*** para ***INTERSECT*** ou ***LEFT JOIN*** com ***WHERE IS NULL*** para ***EXCEPT***) ou combinações de ***UNION*** e filtros para obter o mesmo resultado.

7.3. Prática no MySQL Workbench

Agora que compreendemos a teoria, vamos aplicar esses conhecimentos na prática utilizando o MySQL Workbench. Abriremos o ambiente e executaremos os comandos para visualizar os resultados.

7.3.1. Abrindo o MySQL Workbench

1. Abra o MySQL Workbench.
2. Clique em "Local MySQL" (ou sua conexão local configurada).
3. Com o banco de dados aberto e suas tabelas e registros visíveis, clique no campo da "Query 1" para começar a digitar os comandos.

7.3.2. Comandos de Subconsultas

7.3.2.1. Subconsulta no WHERE com IN

Objetivo: Encontrar os nomes de todos os

alunos que estão matriculados no curso de 'Matemática'.

sql

```sql
SELECT nome
FROM alunos
WHERE id IN (SELECT aluno_id FROM matriculas
WHERE curso = 'Matemática');
```

Explicação: Esta consulta utiliza uma subconsulta para obter os ***aluno_id***s dos estudantes de Matemática e, em seguida, a consulta principal filtra os nomes dos alunos com base nesses IDs. O resultado mostrará apenas os alunos que fazem parte do curso de Matemática.

7.3.2.2. Subconsulta com EXISTS

Objetivo: Listar os nomes dos alunos para os quais existe pelo menos uma matrícula no curso de 'Física'.

sql

```sql
SELECT nome
FROM alunos a
WHERE EXISTS (SELECT 1 FROM matriculas m
WHERE m.aluno_id = a.id AND m.curso =
'Física');
```

Explicação: O ***EXISTS*** verifica a presença de matrículas em 'Física' para cada aluno. Se houver, o nome do aluno é retornado. É uma forma eficiente de verificar a existência sem precisar retornar os dados da subconsulta.

7.3.2.3. Subconsulta Escalar no SELECT

Objetivo: Mostrar o nome de cada aluno e a contagem total de matrículas de cada um.

sql

```sql
SELECT a.nome,
       (SELECT COUNT(*)
        FROM matriculas m
        WHERE m.aluno_id = a.id) AS
qtd_matriculas
FROM alunos a;
```

Explicação: Para cada aluno, a subconsulta conta quantas matrículas ele possui. O resultado é uma tabela com o nome do aluno e uma nova coluna ***qtd_matriculas*** indicando o total.

7.3.2.4. Subconsulta no FROM (Tabela Derivada)

Objetivo: Consultar apenas os alunos com mais de uma matrícula.

sql

```sql
SELECT t.aluno_id, t.qtd
FROM (
    SELECT aluno_id, COUNT(*) AS qtd
    FROM matriculas
    GROUP BY aluno_id
) AS t
WHERE t.qtd > 1;
```

Explicação: A subconsulta cria uma tabela temporária (***t***) que resume a quantidade de matrículas por aluno. A consulta externa então filtra essa tabela para mostrar apenas os alunos que têm mais de uma matrícula.

7.3.2.5. Subconsulta Correlacionada

Objetivo: Verificar se o aluno tem nota acima da média dos alunos com o mesmo status (ativo ou inativo).

sql

```sql
SELECT a.nome, a.nota
FROM alunos a
WHERE a.nota > (
    SELECT AVG(nota)
    FROM alunos
    WHERE ativo = a.ativo
);
```

Explicação: Esta subconsulta compara a nota de cada aluno com a média das notas dos alunos que compartilham o mesmo status de ***ativo***. A subconsulta é correlacionada porque depende do valor ***a.ativo*** da consulta externa, sendo reexecutada para cada linha.

7.3.3. Comandos de Operadores de Conjunto

7.3.3.1. UNION (Remove Duplicatas)

Objetivo: Criar uma lista única com o nome de todos os alunos que têm nota maior ou igual a

9 OU que estão ativos.

```sql
SELECT nome FROM alunos WHERE nota >= 9
UNION
SELECT nome FROM alunos WHERE ativo = TRUE;
```

Explicação: Este comando combina os nomes dos alunos que atendem a uma das duas condições. Se um aluno satisfizer ambas as condições, seu nome aparecerá apenas uma vez no resultado final, pois o ***UNION*** remove duplicatas.

7.3.3.2. UNION ALL (Mantém Duplicatas)

Objetivo: Criar uma lista com o nome de todos os alunos que têm nota maior ou igual a 9 OU que estão ativos, mantendo duplicatas.

```sql
SELECT nome FROM alunos WHERE nota >= 9
UNION ALL
SELECT nome FROM alunos WHERE ativo = TRUE;
```

Explicação: Similar ao ***UNION***, mas se um aluno aparecer em ambas as listas (nota >= 9 e ativo = TRUE), seu nome será listado duas vezes no resultado, pois o ***UNION ALL*** não remove duplicatas. Isso pode ser útil para contagens ou análises onde a frequência de ocorrência é importante.

7.4. Exercício prático de Subconsultas e conjuntos

Nesta atividade, vamos aplicar na prática os conceitos de subconsultas (subqueries) e operadores de conjuntos (***UNION***). Você irá criar um novo banco de dados, duas tabelas, inserir dados e, em seguida, resolver uma série de desafios para extrair informações complexas, exatamente como faria em um ambiente de trabalho real.

7.4.1. Preparação do Ambiente

Primeiro, vamos criar nosso espaço de trabalho. Abra o MySQL Workbench e execute os seguintes comandos para criar o banco de dados e as tabelas que usaremos.

7.4.1.1. Crie e selecione o banco de dados:

```sql
CREATE DATABASE apostila7;
USE apostila7;
```

Dica: O comando ***USE*** é fundamental para garantir que todas as tabelas e dados sejam criados no lugar certo! Certifique-se de usar para utilizar todos os comandos neste banco de dados.

7.4.1.2. Crie a tabela funcionarios:

Esta tabela guardará informações sobre os colaboradores de uma empresa.

```sql
CREATE TABLE funcionarios (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100),
    cargo VARCHAR(50),
    salario DECIMAL(10, 2)
);
```

7.4.1.3. Crie a tabela projetos_alocados:

Esta tabela registrará em quais projetos cada funcionário está trabalhando.

```sql
CREATE TABLE projetos_alocados (
    id_alocacao INT PRIMARY KEY
AUTO_INCREMENT,
    id_funcionario INT,
    nome_projeto VARCHAR(100),
    FOREIGN KEY (id_funcionario) REFERENCES
funcionarios(id)
);
```

Observação: Note o uso da ***FOREIGN KEY*** (Chave Estrangeira). Ela cria uma ligação entre as duas tabelas, garantindo que só possamos alocar funcionários que realmente existem na tabela ***funcionarios***.

7.4.2. Inserindo os Dados

Agora que temos as tabelas, vamos populá-las com dados. Execute os comandos *INSERT* abaixo.

7.4.2.1. Insira os dados na tabela funcionarios:

sql

```sql
INSERT INTO funcionarios (nome, cargo,
salario) VALUES
('Ana Silva', 'Desenvolvedor Sênior',
7500.00),
('Bruno Costa', 'Gerente de Projetos',
9200.00),
('Carla Dias', 'Analista de BI', 6100.00),
('David Rocha', 'Desenvolvedor Pleno',
5800.00),
('Eliana Matos', 'Analista de BI', 6100.00),
('Fábio Borges', 'Desenvolvedor Sênior',
7500.00);
```

7.4.2.2. Insira os dados na tabela projetos_alocados:

sql

```sql
INSERT INTO projetos_alocados
(id_funcionario, nome_projeto) VALUES
(1, 'Sistema Phoenix'),
(1, 'App Inova'),
(2, 'Sistema Phoenix'),
(3, 'Dashboard Analytics'),
(4, 'App Inova'),
(6, 'Sistema Phoenix');
```

Análise Rápida: Observe que a funcionária Ana (ID 1) está em dois projetos. O funcionário David (ID 4) está em um projeto, e a Eliana (ID 5) não foi alocada a nenhum projeto ainda. Essas variações serão importantes para nossas consultas!

7.4.3. Hora do Desafio - Realizando as Consultas

Com tudo pronto, é hora de aplicar o que você aprendeu! Resolva os desafios abaixo.

7.4.3.1. Desafio 1: Subconsulta no WHERE com IN

Encontre os nomes de todos os funcionários que estão alocados no projeto chamado '**Sistema Phoenix**'.

Dica: Você precisa de uma subconsulta que retorne a lista de ***id_funcionario*** que trabalham nesse projeto. A consulta principal usará essa lista para filtrar os nomes na tabela ***funcionarios***.

7.4.3.2. Desafio 2: Subconsulta Escalar no SELECT

Liste o nome de cada funcionário e, ao lado, uma coluna chamada ***quantidade_projetos*** que mostre em quantos projetos ele está alocado.

Dica: Para cada linha da tabela ***funcionarios***, você precisa executar uma subconsulta que conte (*COUNT*) as alocações daquele funcionário na tabela ***projetos_alocados***.

7.4.3.3. Desafio 3: Subconsulta no FROM (Tabela Derivada)

Mostre o ***nome_projeto*** e a ***total_funcionarios*** apenas para os projetos que têm mais de 2 funcionários alocados.

Dica: Primeiro, crie uma subconsulta que agrupe os projetos e conte quantos funcionários há em cada um. Depois, use essa subconsulta como uma tabela temporária na sua consulta principal para filtrar os resultados.

7.4.3.4. Desafio 4: Conjuntos com UNION

Crie uma lista única com o nome de todos os funcionários que são '**Desenvolvedor Sênior**' OU que estão alocados no projeto '**App Inova**'. A lista não deve conter nomes repetidos.

Dica: Crie a primeira consulta para selecionar os desenvolvedores seniores. Crie a segunda consulta (usando uma subconsulta com *IN*) para selecionar os funcionários do 'App Inova'. Junte as duas com ***UNION***.

Banco de Dados com MySQL

Banco de dados com MySql

8. Alterações DDL

Aula 8

Olá, seja bem-vindo à nossa oitava e última aula de banco de dados com MySQL! Nesta aula, vamos entender e aprender sobre as **Alterações DDL (Data Definition Language)** dentro do banco de dados, utilizando o MySQL Workbench.

8.1. O que são Alterações DDL?

Pense no seu banco de dados como uma casa: as tabelas são os cômodos e as colunas são os móveis. As **Alterações DDL** são as ferramentas e obras que mudam a própria estrutura da casa. Imagine trocar uma porta de lugar, colocar uma nova prateleira, derrubar uma parede ou pintar de outra cor. Em SQL, essas "obras" são comandos que mudam como os dados serão armazenados, não os dados em si (ou pelo menos não sempre).

8.1.1. Data Definition Language (DDL)

DDL é a parte do SQL usada para definir e alterar a estrutura do banco de dados. Os comandos principais que você já deve ter conhecido são:

- ***CREATE***: Para criar bancos de dados, tabelas, etc.

- ***DROP:*** Para apagar tabelas, bancos de dados, etc.

- ***ALTER:*** Para alterar o que já existe (tabelas, colunas, etc.).

- ***TRUNCATE:*** Para apagar todo o conteúdo de uma tabela, mantendo sua estrutura.

Quando falamos em "alterações DDL", estamos nos referindo a tudo que envolve ***ALTER TABLE*** e operações similares para modificar tabelas, colunas, índices, chaves, entre outros elementos estruturais.

8.1.2. Por que usar ALTER e fazer mudanças DDL?

Você utiliza DDL quando a estrutura do banco de dados precisa ser modificada devido a mudanças nos requisitos. Por exemplo:

- Adicionar um novo campo (coluna) para armazenar um telefone.
- Mudar o tipo de uma coluna de ***VARCHAR(50)*** para ***VARCHAR(100)***.
- Criar um índice para otimizar a velocidade das buscas.
- Adicionar uma chave estrangeira para garantir a integridade referencial entre tabelas.

8.2. Principais Tipos de Alteração DDL

Vamos entender os principais tipos de alteração DDL, utilizando exemplos de comandos e explicando suas funcionalidades.

8.2.1. Adicionar uma Coluna

Este tipo de alteração serve para adicionar uma nova coluna a uma tabela existente.

Exemplo de Comando:

sql
```sql
ALTER TABLE alunos ADD COLUMN telefone
VARCHAR(20) AFTER nome;
```

Este comando cria a coluna ***telefone*** do tipo texto, com até 20 caracteres. ***O AFTER nome*** indica a posição visual da nova coluna, mas é um comando opcional. **Cuidado:** Em tabelas com centenas de milhares de linhas, o banco pode travar a tabela enquanto realiza a alteração, sendo fundamental ter cautela em ambientes de produção.

8.2.2. Remover uma Coluna

Serve para remover uma coluna que não está mais sendo utilizada no banco de dados.

Exemplo de Comando:

```sql
ALTER TABLE alunos DROP COLUMN telefone;
```

Este comando elimina a coluna telefone e todos os dados que estavam nela. É uma operação destrutiva e sem "lixeira", portanto, faça backup se precisar dos dados antes de executar.

8.2.3. Modificar o Tipo ou Definição de uma Coluna

Permite modificar uma coluna existente ou até mesmo renomeá-la.

Exemplos de Comandos:

```sql
ALTER TABLE alunos
MODIFY COLUMN nome VARCHAR(150) NOT NULL;

ALTER TABLE alunos
CHANGE COLUMN nome nome_completo
VARCHAR(150) NOT NULL;
```

- O comando MODIFY altera o tipo/atributos de uma coluna, mantendo o mesmo nome.
- O comando CHANGE permite renomear a coluna e alterar seu tipo e atributos simultaneamente. No exemplo, nome foi renomeado para nome_completo.

Cuidado: Se houver dados incompatíveis (por exemplo, mudar VARCHAR para INT), a alteração pode falhar ou truncar dados.

8.2.4. Entendendo TRUNCATE: A História do Caderno

Antes de continuarmos com os tipos de Alterações DDL, vamos entender o que significa ***TRUNCATE***.

Imagine que uma tabela do seu banco de dados é um caderno cheio de anotações. Você tem duas maneiras de apagar tudo:

1. **Apagar página por página (Comando *DELETE*):** Você pode pegar uma borracha e apagar cada anotação, uma por uma, até não sobrar mais nada. É um processo lento e cuidadoso, e você pode escolher quais anotações apagar.
2. **Arrancar todas as folhas de uma vez (Comando *TRUNCATE*)**: Você pode simplesmente arrancar todas as páginas do caderno de uma só vez, deixando apenas a capa e a estrutura. É um processo instantâneo e apaga tudo de uma vez.

O comando ***TRUNCATE TABLE*** é usado para remover todos os registros (linhas) de uma tabela de forma muito rápida. Ele age como um "botão de reset" para os dados da tabela, deixando a estrutura dela (as colunas, os tipos de dados, etc.) completamente intacta, pronta para receber novos dados.

Diferenças entre TRUNCATE e DELETE:

- ***DELETE*** pode deletar dados específicos usando a cláusula ***WHERE***, enquanto ***TRUNCATE*** apaga tudo.
- ***TRUNCATE*** é geralmente mais rápido para apagar todos os dados.
- ***TRUNCATE*** reseta o contador de ***AUTO_INCREMENT*** (se houver), fazendo os novos registros contarem do 1 novamente.

8.3. Adicionar / Remover Chave Primária

Serve para adicionar ou remover uma chave primária a uma tabela, definindo o identificador único do registro.

Exemplos de Comandos:

```sql
ALTER TABLE alunos
ADD PRIMARY KEY (id);

-- Para remover:
ALTER TABLE alunos
DROP PRIMARY KEY;
```

A chave primária garante a unicidade de cada registro e é fundamental para o estabelecimento de relacionamentos entre tabelas. Removê-la pode ter implicações sérias para índices e chaves estrangeiras (Foreign Keys).

Observação: Se a coluna ***ID*** já possui uma chave primária ou está definida como ***AUTO_INCREMENT***, tentar adicionar uma nova chave primária resultará em erro, pois o MySQL impede a remoção de uma chave primária que é essencial para o funcionamento do ***AUTO_INCREMENT***.

8.3.1. Adicionar / Remover Índice (Index)

Serve para criar atalhos que agilizam as consultas no banco de dados.

Exemplos de Comandos:

```sql
ALTER TABLE alunos
ADD INDEX idx_nome (nome);

-- Para remover:
ALTER TABLE alunos
DROP INDEX idx_nome;
```

Os **índices** melhoram significativamente a velocidade de leitura e consulta por coluna, funcionando como o índice de um livro. No entanto, eles consomem espaço e podem degradar o desempenho de operações de inserção e atualização. Crie índices apenas nas colunas frequentemente usadas em cláusulas ***WHERE***, ***JOIN*** ou ***ORDER BY.***

Como visualizar índices:

- No MySQL Workbench, clique duas vezes na tabela e expanda a seção "Indexes".
- Ou usando o comando SQL: SHOW INDEX FROM alunos;

8.3.2. Adicionar / Remover Chave Estrangeira (Foreign Key)

Serve para adicionar ou remover uma chave estrangeira, ligando tabelas entre si com regras de integridade referencial.

Exemplo de Comando:

```sql
ALTER TABLE matriculas
ADD CONSTRAINT fk_aluno
  FOREIGN KEY (aluno_id) REFERENCES
alunos(id)
  ON DELETE CASCADE;
```

Este comando define que ***matriculas.aluno_id*** deve existir em ***alunos.id***, estabelecendo uma ligação entre as tabelas. O ***ON DELETE CASCADE*** significa que, se um aluno for apagado da tabela ***alunos***, todas as suas matrículas correspondentes na tabela ***matriculas*** também serão automaticamente excluídas.

Como visualizar chaves estrangeiras:

- No MySQL Workbench, clique duas vezes na tabela e expanda a seção "Foreign Keys".

8.3.3. Renomear Tabela

Este tipo serve apenas para renomear uma tabela existente.

Exemplos de Comandos:

```sql
RENAME TABLE alunos TO estudantes;
-- ou
ALTER TABLE alunos RENAME TO estudantes;
```

Ambos os comandos alteram o nome da tabela ***alunos*** para ***estudantes***. O conteúdo interno da tabela permanece o mesmo, apenas o nome é modificado. É importante atualizar

qualquer código ou query que utilize o nome antigo da tabela.

8.3.4. Trocar Engine, Charset ou Collation

Este tipo serve para mudar a "base" ou a forma de guardar e ordenar texto na tabela.

Exemplos de Comandos:

sql

```sql
ALTER TABLE alunos ENGINE = InnoDB;
ALTER TABLE alunos CONVERT TO CHARACTER SET
utf8mb4 COLLATE utf8mb4_unicode_ci;
```

- O comando ***ENGINE*** define o mecanismo de armazenamento da tabela. ***InnoDB*** é o mais utilizado, oferecendo suporte a transações e chaves estrangeiras.
- ***CHARACTER SET*** e ***COLLATE*** controlam como o texto é armazenado e ordenado (importante para acentos, emojis e ordenação). ***utf8mb4_unicode_ci*** é a escolha recomendada para bancos de dados modernos, pois suporta uma ampla gama de caracteres.

8.3.5. Definir/Alterar DEFAULT, NOT NULL, UNIQUE

Serve para impor regras e restrições nas tabelas, garantindo a integridade dos dados.

Exemplos de Comandos:

sql

```sql
ALTER TABLE alunos ALTER COLUMN ativo SET
DEFAULT TRUE;
ALTER TABLE alunos MODIFY COLUMN email
VARCHAR(150) UNIQUE;
```

- ***DEFAULT:*** Define um valor automático para a coluna caso nenhum valor seja informado pelo usuário (ex: todo novo aluno começa com ***ativo = TRUE***).
- ***NOT NULL:*** Impede que a coluna aceite valores nulos.
- ***UNIQUE:*** Garante que não haja valores duplicados na coluna (útil para e-mails ou CPFs). **Cuidado**: Se já existirem valores repetidos na coluna, o MySQL não permitirá criar a restrição ***UNIQUE*** até que você os corrija.

8.4. Efeitos Colaterais e Cuidados ao Utilizar DDL

É crucial ter em mente os seguintes pontos ao realizar operações DDL:

- **Operações podem bloquear tabelas:** ***ALTER TABLE*** em tabelas grandes frequentemente bloqueia operações de leitura/escrita enquanto a estrutura é alterada. Em ambientes de produção, planeje janelas de manutenção ou utilize ferramentas/versões do MySQL que suportem alterações online (como o InnoDB/online DDL).
- **Backup antes de mudanças importantes:** Sempre faça backup antes de remover colunas, chaves ou tabelas. Isso permite a recuperação em caso de erros.
- **Perda de dados:** Comandos como ***DROP COLUMN*** apagam dados permanentemente; alterar o tipo de uma coluna pode truncar ou incompatibilizar dados. Verifique sempre antes de executar.
- **Transações e DDL:** Operações DDL normalmente realizam ***commit*** automático e não podem ser revertidas com ***ROLLBACK*** após a execução de ***ALTER TABLE***.

8.5. ROLLBACK e Transações no MySQL

Para entender o ***ROLLBACK***, precisamos primeiro conhecer o conceito de Transação ***(TRANSACTION)***.

8.5.1. A História do Checkpoint no Videogame

Pense no ***ROLLBACK*** como o "checkpoint" de um videogame. Imagine que você está em uma

fase difícil, passa por vários desafios, coleta itens e, de repente, comete um erro e perde. O jogo não acaba ali; ele te leva de volta para o último checkpoint que você salvou, como se nada daquele trecho tivesse acontecido. Todo o progresso depois do checkpoint é desfeito. O ***ROLLBACK*** funciona de maneira similar para o seu banco de dados.

8.5.2. Transações (TRANSACTION)

Uma **Transação** é um conjunto de comandos SQL que são tratados como um único bloco de trabalho, uma única operação. A regra de uma transação é: "ou tudo funciona, ou nada funciona".

Exemplo da Transferência Bancária:

Pense em uma transferência bancária. Duas coisas precisam acontecer: o dinheiro sai da sua conta e, em seguida, o dinheiro entra na conta de outra pessoa. Se o primeiro passo acontecer, mas o sistema falhar antes do segundo, o dinheiro simplesmente sumiria! Isso seria um desastre. Uma transação garante que, se qualquer parte falhar, a operação inteira é desfeita, retornando ao estado inicial.

8.5.3. Comandos de Transação

Neste processo, existem três comandos importantes:

1. ***START TRANSACTION;***: Inicia o "modo de segurança". É como dizer ao MySQL: "Ok, a partir de agora, não salve nada permanentemente até eu mandar. Fique de olho no que estou fazendo". Este é o seu checkpoint.
2. ***COMMIT;***: Salva permanentemente todas as alterações que você fez desde o ***START TRANSACTION***. É como chegar no próximo ponto de salvamento do jogo e gravar seu progresso. Após o ***COMMIT***, não é possível voltar atrás com um ***ROLLBACK***.
3. ***ROLLBACK;***: Desfaz todas as alterações feitas desde o ***START TRANSACTION***. É o comando que você usa quando algo deu errado ou quando você simplesmente desiste da operação. Ele restaura o banco de dados para o estado em que estava no momento do seu "checkpoint".

8.6. Prática no MySQL Workbench

Vamos aplicar os comandos de alteração DDL na prática dentro do MySQL Workbench.

8.6.1. Adicionar uma Coluna

Comando:

sql

```sql
ALTER TABLE alunos ADD COLUMN telefone
VARCHAR(20) AFTER nome;
```

Este comando adiciona uma nova coluna chamada ***telefone*** à tabela ***alunos***, do tipo texto (***VARCHAR***) com até 20 caracteres, posicionada logo após a coluna ***nome***. O ***AFTER nome*** é opcional, mas ajuda a organizar a ordem das colunas visualmente. Em tabelas com muitos registros, essa operação pode demorar ou travar momentaneamente, pois o MySQL precisa reescrever a tabela inteira.

8.6.2. Remover uma Coluna

Comando:

sql

```sql
ALTER TABLE alunos DROP COLUMN telefone;
```

Este comando remove completamente a coluna ***telefone*** da tabela ***alunos***. Ao remover, todos os dados que estavam nessa coluna também serão apagados.

8.6.3. Modificar ou Renomear uma Coluna

Comandos:

```sql
ALTER TABLE alunos MODIFY COLUMN nome
VARCHAR(150) NOT NULL;
ALTER TABLE alunos CHANGE COLUMN nome
nome_completo VARCHAR(150) NOT NULL;
```

- O comando ***MODIFY*** altera o tipo ou restrições de uma coluna existente. No exemplo, o tamanho do campo ***nome*** é aumentado para 150 caracteres e a restrição ***NOT NULL*** é aplicada.
- O comando ***CHANGE*** permite renomear a coluna e alterar suas propriedades ao mesmo tempo. No exemplo, ***nome*** foi renomeado para ***nome_completo***.

8.6.4. TRUNCATE (Apagar Todos os Dados da Tabela)

Comando:

```sql
TRUNCATE TABLE matriculas;
```

Este comando remove todos os registros da tabela ***matriculas***, mas mantém sua estrutura intacta. Lembre-se, ***TRUNCATE*** não permite ***WHERE*** e não pode ser revertido com ***ROLLBACK***.

8.6.5. Adicionar / Remover Chave Primária

Comandos:

```sql
ALTER TABLE alunos ADD PRIMARY KEY (id);
-- Para remover:
ALTER TABLE alunos DROP PRIMARY KEY;
```

Ao tentar adicionar uma chave primária em uma coluna que já a possui ou que é ***AUTO_INCREMENT***, ocorrerá um erro, pois o MySQL impede a remoção de uma chave primária que é essencial para o funcionamento do ***AUTO_INCREMEN***T.

8.6.6. Adicionar / Remover Índice (Index)

Comandos:

```sql
ALTER TABLE alunos ADD INDEX idx_nome
(nome_completo);
-- Para remover:
ALTER TABLE alunos DROP INDEX idx_nome;
```

Adicionar um índice na coluna ***nome_completo*** (ou ***nome*** se renomeada) agiliza buscas. Lembre-se de usar índices com cautela, pois muitos índices podem lentificar inserções e atualizações.

8.6.7. Adicionar Chave Estrangeira (Foreign Key)

Comando:

```sql
ALTER TABLE matriculas
ADD CONSTRAINT fk_aluno
  FOREIGN KEY (aluno_id) REFERENCES
alunos(id)
  ON DELETE CASCADE;
```

Este comando cria uma ligação entre ***matriculas.aluno_id*** e ***alunos.id***. O ***ON DELETE CASCADE*** garante que, ao excluir um aluno, suas matrículas relacionadas também sejam excluídas.

8.6.8. Renomear Tabela

Comandos:

```sql
RENAME TABLE alunos TO estudantes;
-- ou
ALTER TABLE alunos RENAME TO estudantes;
```

Estes comandos alteram o nome da tabela. Para fins desta aula, manteremos o nome ***alunos***.

8.6.9. Trocar Engine, Charset e Collation

Comandos:

sql

```sql
ALTER TABLE alunos ENGINE = InnoDB;
ALTER TABLE alunos CONVERT TO CHARACTER SET
utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Estes comandos permitem configurar o mecanismo de armazenamento e a forma como o texto é tratado. Para esta aula, presumimos que as tabelas já estão configuradas adequadamente.

8.6.10. Definir/Alterar DEFAULT, NOT NULL, UNIQUE

Comandos:

sql

```sql
ALTER TABLE alunos ALTER COLUMN ativo SET
DEFAULT TRUE;
ALTER TABLE alunos MODIFY COLUMN email
VARCHAR(150) UNIQUE;
```

O ***DEFAULT*** define um valor padrão para uma coluna. O ***UNIQUE*** garante que não haja valores duplicados. Um erro pode ocorrer se a coluna ***email*** não existir ou já contiver valores duplicados antes de aplicar a restrição ***UNIQUE***.

8.7. Atividade Prática

Para consolidar o aprendizado, realize as seguintes atividades no MySQL Workbench, utilizando a estrutura do banco de dados ***curso*** com as tabelas ***alunos*** e ***matriculas***:

1. **Adicionar Coluna:** Crie um comando que adicione uma nova coluna chamada ***telefone*** à tabela ***alunos***, com tipo ***VARCHAR(20***), logo após a coluna ***nome_completo***.
2. **Remover Coluna**: Faça um comando que remova a coluna ***telefone*** que você acabou de criar da tabela ***alunos***.
3. **Modificar Coluna:** Faça um comando que aumente o tamanho máximo permitido da coluna ***nome_completo*** para 150 caracteres e garanta que ela não aceite valores nulos.
4. **Renomear Coluna:** Renomeie a coluna ***nome_completo*** para nome, mantendo o tipo ***VARCHAR(150)*** e a restrição ***NOT NULL***.
5. **Apagar Registros (TRUNCATE):** Faça um comando que apague todos os registros da tabela ***matriculas***, mas mantenha a estrutura da tabela.

Anotações