Dominando o SQL: Fundamentos e Gerenciamento de Dados

Bem-vindos à nossa imersão no mundo do SQL! Nesta apresentação, exploraremos os pilares da linguagem de consulta estruturada, desde seus conceitos fundamentais até técnicas avançadas de manipulação e modelagem de dados. Baseado nos ensinamentos de "Introdução à Linguagem SQL" de Thomas Nield, nosso objetivo é fornecer uma compreensão sólida e prática que capacite tanto estudantes quanto profissionais de TI a otimizar suas operações com bancos de dados. Prepare-se para desvendar o poder do SQL e transformar a maneira como você interage com os dados.

1. Fundamentos de Bancos de Dados Relacionais

Definição e Propósito

Um banco de dados, em sua essência, é uma ferramenta para "coletar e organizar dados". No contexto profissional, referimo-nos a um Sistema de Gerenciamento de Banco de Dados Relacional (RDBMS), que contém "uma ou mais tabelas que podem estar relacionadas entre si". Essa estrutura relacional é o cerne da sua eficiência, permitindo organização e acesso flexível à informação.

Estrutura e Relacionamentos

- **Tabelas:** Similares a planilhas, com colunas e linhas, organizam os dados de forma estruturada.
- **Relacionamentos:** A força dos RDBMS reside na capacidade de "tabelas se relacionarem umas com as outras". Isso permite que dados sejam buscados e combinados de diferentes fontes, como associar um `CUSTOMER_ID` em uma tabela de pedidos aos detalhes do cliente na tabela `CUSTOMER`.

Normalização e Vantagens

- **Normalização:** É o processo de "separar diferentes tipos de dados em suas próprias tabelas", um passo crucial para evitar a "redundância". A redundância não apenas ocupa "espaço de armazenamento desnecessário", mas também dificulta a manutenção e aumenta o risco de inconsistências.
- **Benefícios:** Com tabelas normalizadas, qualquer alteração em um dado precisa ser feita em "apenas um único registro" na tabela apropriada, garantindo a integridade. O operador `JOIN` (Capítulo 8) é então usado para "mesclar essas tabelas em uma consulta para visualizar as informações de forma combinada", recuperando a visão completa dos dados.

Soluções de Banco de Dados

Existem duas categorias principais, cada uma com suas aplicações:

- **Bancos de dados leves:** Projetados para usuários únicos ou pequenos grupos, com "pouco ou nenhum overhead". Exemplos incluem SQLite ("gratuito, leve e fácil de usar", ideal para aprendizado e sistemas embarcados) e Microsoft Access.
- **Bancos de dados centralizados:** Suportam "dezenas, centenas ou milhares de usuários e aplicativos simultaneamente", executados em um servidor. Incluem gigantes como MySQL (open source e amplamente usado), Microsoft SQL Server, Oracle e PostgreSQL.

> **A Linguagem SQL é Universal:** A grande vantagem é que a experiência de uso do SQL é "bastante uniforme" entre as diferentes plataformas. O conhecimento adquirido com um banco de dados leve como SQLite é totalmente aplicável em sistemas de grande escala, tornando o aprendizado altamente transferível.

2. Recuperação de Dados com SELECT

A Instrução SELECT: A Porta de Entrada para Seus Dados

A instrução `SELECT` é a "**operação mais comum**" no SQL, sendo a principal forma de "solicitar dados de uma ou mais tabelas e exibi-los". É o ponto de partida para qualquer análise ou visualização de dados.

- `SELECT * FROM [nome_da_tabela];`: Utilize o asterisco (`*`) como "**curinga**" para selecionar **todas** as colunas de uma tabela. Ideal para uma visão geral rápida.
- `SELECT Coluna1, Coluna2 FROM [nome_da_tabela];`: Para maior especificidade e otimização, selecione **colunas específicas** pelo nome. Isso melhora a performance e a legibilidade da consulta.

Expressões, Aliases e Funções: Refinando seus Resultados

- **Cálculos Dinâmicos:** A instrução `SELECT` permite "efetuar cálculos em uma ou mais colunas" diretamente na consulta. Por exemplo, `PRICE * 1.07` pode calcular um preço com imposto. É importante notar que essas novas colunas são "**calculadas dinamicamente**" e não são armazenadas fisicamente na tabela.
- **Aliases (`AS`):** Atribua um nome temporário e mais legível a um valor calculado ou a uma coluna usando a palavra-chave `AS`. Exemplo: `PRICE * 1.07 AS TAXED_PRICE`. Recomenda-se usar underscore (`_`) em vez de espaços em nomes de aliases para evitar problemas de compatibilidade ou a necessidade de aspas.
- **Funções:** O SQL possui funções internas poderosas, como `round()`, para formatar resultados (ex: `round(PRICE * 1.07, 2)` para arredondar para duas casas decimais).
- **Operadores Matemáticos:** Utilize os operadores básicos (`+`, `-`, `*`, `/`) e o operador de módulo (`%`) para diversas operações aritméticas.

Concatenação de Texto: Combinando Informações

- **Operador `||`:** É o operador padrão no SQL para "mesclar dois ou mais dados, tratando-os como texto". É extremamente útil para construir strings combinadas. Exemplo: `CITY || ', ' || STATE AS LOCATION` criaria uma coluna com "São Paulo, SP". Uma grande vantagem é que ele **funciona com qualquer tipo de dado**, que é implicitamente convertido para texto.
- **Função `CONCAT()`:** Em algumas plataformas, como MySQL, a função `CONCAT()` é utilizada para o mesmo propósito, permitindo a concatenação de múltiplos argumentos.

3. Filtragem de Dados com WHERE

O Propósito da Cláusula WHERE

A cláusula `WHERE` é a espinha dorsal da seletividade em consultas SQL. Sua função primordial é "filtrar registros" de uma consulta `SELECT`, garantindo que apenas aqueles que "atendem a um ou mais critérios especificados" sejam exibidos no conjunto de resultados. É aqui que você define as condições para refinar seus dados.

Filtragem Numérica e Booleana

- **Operadores de Comparação:** Utilize operadores como `=` (igual a), `!=` ou `<>` (desigual a), `>` (maior que), `<` (menor que), `>=` (maior ou igual) e `<=` (menor ou igual). A cláusula `BETWEEN` é especialmente útil para intervalos inclusivos, por exemplo, `WHERE id BETWEEN 100 AND 200`.
- **Valores Booleanos:** Em muitos bancos de dados, valores booleanos são representados numericamente (tipicamente 1 para verdadeiro e 0 para falso). No SQLite, é comum usar `WHERE coluna = 1` ou `WHERE coluna` para verdadeiro, e `WHERE coluna = 0` ou `WHERE NOT coluna` para falso.

Combinando Condições (AND, OR, IN)

- `AND`**:** Exige que "todas as condições sejam verdadeiras" para que o registro seja incluído.
- `OR`**:** Exige que "pelo menos uma das condições seja verdadeira".
- `IN`**:** Oferece uma forma mais elegante e eficiente de listar múltiplos valores para igualdade. Por exemplo, `WHERE MONTH IN (3, 6, 9, 12)` é mais conciso do que múltiplos `OR`. O operador `NOT IN` exclui os valores da lista.
- **Operador de Módulo (`%`):** Embora seu uso primário seja matemático, pode ser empregado para encontrar valores divisíveis (por exemplo, `WHERE ID % 2 = 0` para IDs pares). No Oracle, a função `MOD()` é utilizada.

Filtragem de Texto

- **Aspas Simples:** Literais textuais devem sempre ser inseridos entre aspas simples (`'texto'`).
- `LIKE` **e Curingas:** Permite buscar padrões de texto, oferecendo flexibilidade em comparações de string.
  - `%`: Representa qualquer número de caracteres (inclusive nenhum). Ex: `'A%'` para strings que começam com 'A'.
  - `_` (underscore): Representa um único caractere. Ex: `'_B%'` para strings que têm 'B' na segunda posição.

Manipulando NULL: A Ausência de Valor

`NULL` representa a "ausência completa de valor", e não é o mesmo que zero, uma string vazia (`''`) ou um espaço em branco (`' '`).

- `IS NULL / IS NOT NULL`**:** São os únicos operadores para verificar a presença ou ausência de valores nulos (ex: `WHERE snow_depth IS NULL`).
- **Função** `coalesce()`**:** Pode ser usada para converter um valor nulo em um valor-padrão (ex: `coalesce(precipitation, 0)` retorna 0 se a precipitação for nula).

Agrupando Condições: Precedência e Clareza

É "crucial agrupar as condições relacionadas entre parênteses" para garantir a interpretação correta e evitar ambiguidades. Os parênteses definem a precedência das operações lógicas, assim como na matemática. Sem eles, a ordem de avaliação (`AND` antes de `OR`) pode levar a resultados inesperados.

4. Agregação e Ordenação de Dados

Agregação de Dados: Resumindo Informações

Agregação envolve a "criação de um total a partir de múltiplos registros". Isso é fundamental para obter métricas e insights de grandes volumes de dados.

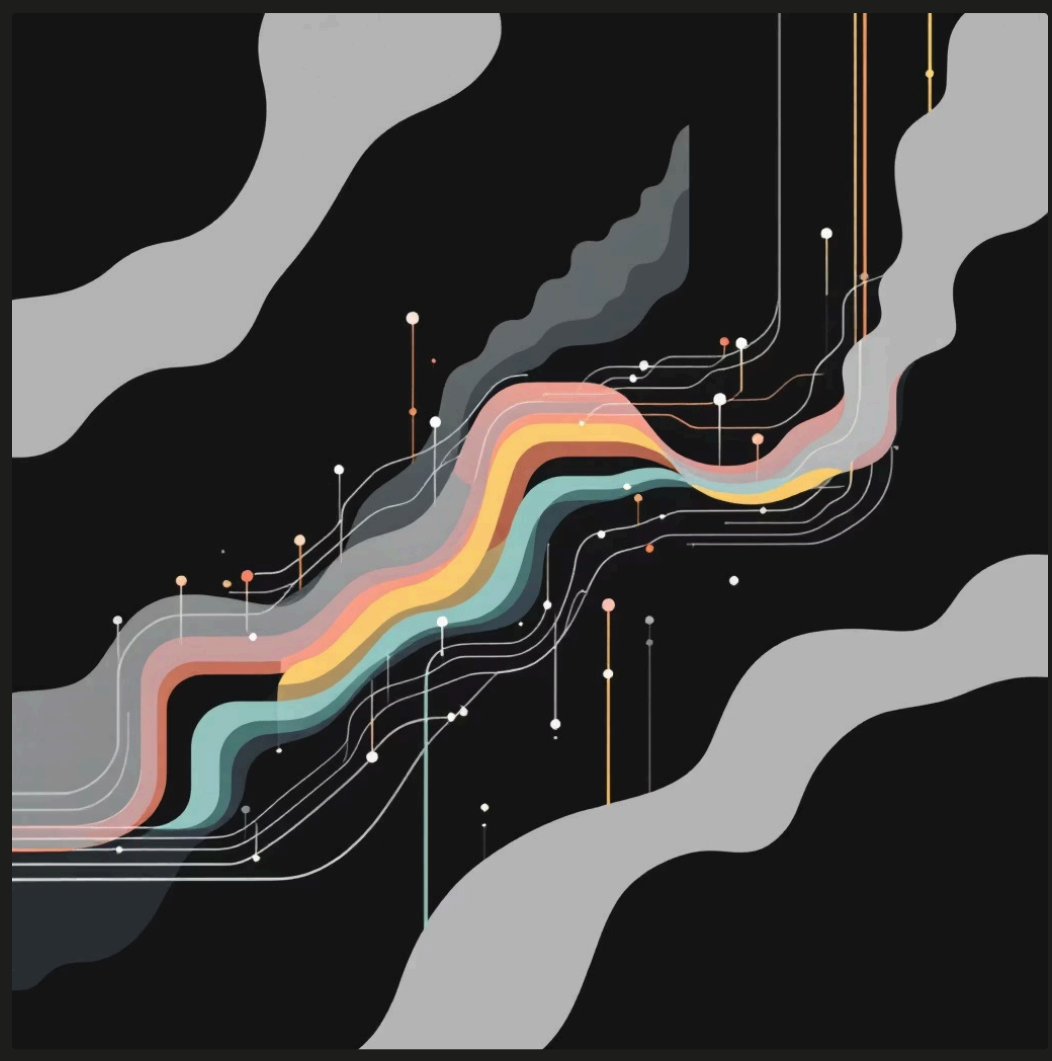
- `COUNT(*)`**:** Conta o número total de registros retornados pela consulta.
- `GROUP BY`**:** Permite "agrupar esses totais por qualquer coluna especificada", controlando o escopo da agregação. Por exemplo, você pode calcular a soma de vendas por região. Embora algumas plataformas suportem o uso de posições ordinais (ex: `GROUP BY 1, 2`), é uma prática menos recomendada pela legibilidade.

Funções de Agregação

As funções de agregação são poderosas e importantes: elas "nunca incluem valores nulos em seus cálculos", o que é crucial para a precisão dos resultados.

- `COUNT(coluna)`**:** Conta o número de valores **não nulos** em uma coluna específica.
- `SUM()`**:** Calcula a soma dos valores numéricos.
- `MIN()`**:** Encontra o valor mínimo.
- `MAX()`**:** Encontra o valor máximo.
- `AVG()`**:** Calcula a média aritmética dos valores.

É possível usar "múltiplas operações de agregação" em uma única consulta, permitindo obter diferentes métricas simultaneamente.



Ordenação de Registros com ORDER BY

A cláusula `ORDER BY` é utilizada para "classificar os resultados da consulta", controlando a ordem de exibição. Ela é inserida após as cláusulas `WHERE` e `GROUP BY`.

- `ASC` **(padrão):** Ordena os resultados em ordem crescente. Pode ser omitido, pois é o comportamento padrão.
- `DESC`**:** Ordena os resultados em ordem decrescente.

Você pode ordenar por uma ou mais colunas, especificando a ordem de prioridade (ex: `ORDER BY sobrenome ASC, nome DESC`).

Filtrando Agregações com HAVING

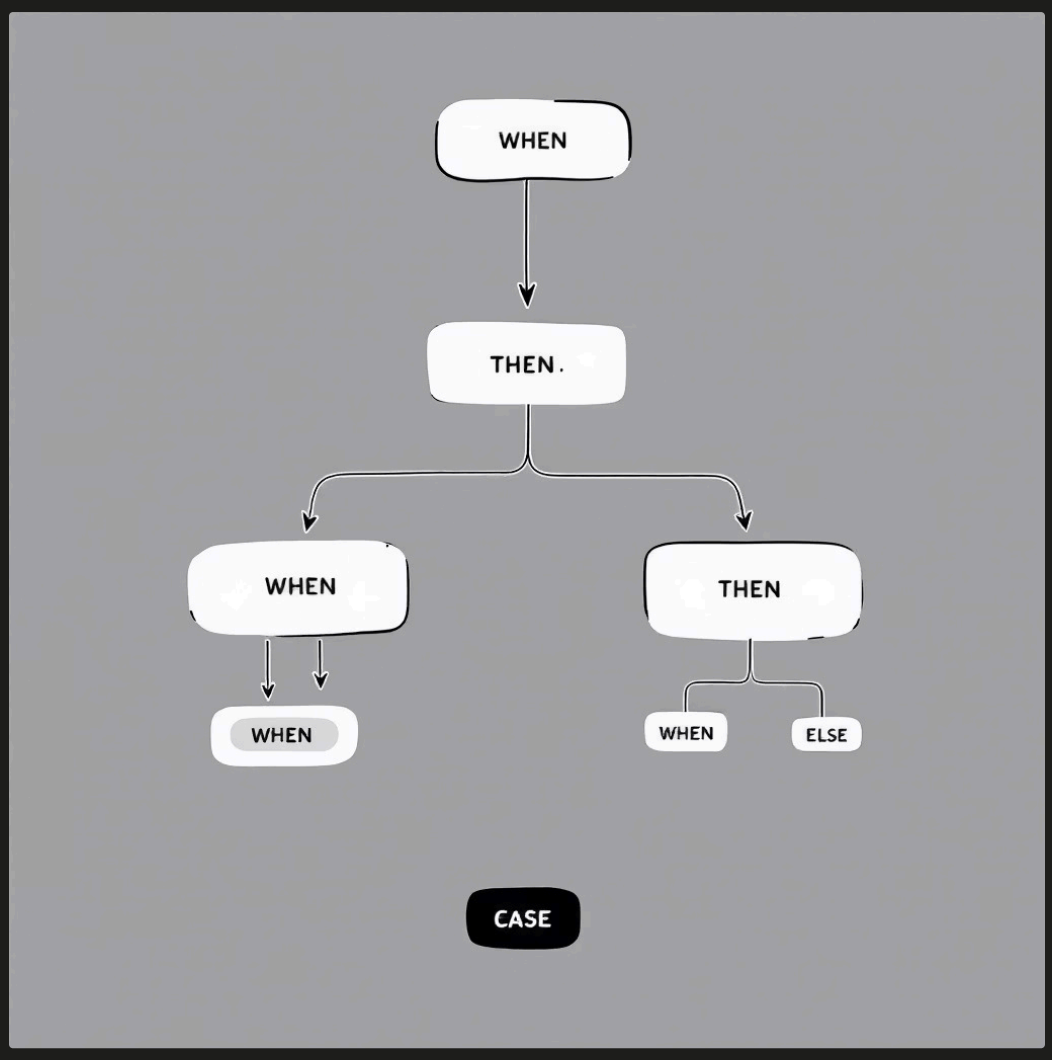
A cláusula `HAVING` é usada para "filtrar registros com base em valores agregados", uma capacidade que `WHERE` não possui.

- **Diferença fundamental:** `WHERE` filtra registros individuais "antes da agregação", enquanto `HAVING` filtra os resultados "depois que as funções de agregação foram calculadas". Por exemplo, você pode usar `HAVING` para mostrar apenas grupos cuja soma de vendas seja superior a um determinado valor.

Registros Distintos com DISTINCT

A palavra-chave `DISTINCT` é usada para "obter um conjunto de resultados sem duplicatas" para uma ou mais colunas. É excelente para listar valores únicos. Exemplo: `SELECT DISTINCT station_number FROM station_data;` retornará cada número de estação apenas uma vez, mesmo que apareça em vários registros.

5. Instruções CASE: Transformação e Agregação Avançada



A Instrução CASE: Lógica Condicional no SQL

O operador CASE é uma ferramenta extremamente versátil que permite "substituir o valor de uma coluna por outro valor" com base em condições específicas. É como uma estrutura IF-THEN-ELSE diretamente na sua consulta SQL.

- **Sintaxe:** Inicia com CASE, utiliza a combinação WHEN [condição] THEN [valor] para cada condição e o valor correspondente. Opcionalmente, pode incluir um ELSE para definir um valor padrão se nenhuma das condições WHEN for atendida. A instrução deve finalizar com END.
- **Avaliação:** O CASE processa as condições de cima para baixo e utiliza "a primeira condição verdadeira encontrada", ignorando todas as subsequentes. Isso é crucial para a lógica do seu código.

```
SELECT
  order_id,
  total_amount,
  CASE
    WHEN total_amount >= 500 THEN 'Grande'
    WHEN total_amount >= 100 THEN 'Médio'
    ELSE 'Pequeno'
  END AS order_size
FROM orders;
```

Agrupando Instruções CASE: Categorização Dinâmica

A verdadeira força do CASE se manifesta quando combinado com GROUP BY. Essa combinação permite realizar "transformações poderosas" e agregar dados com base nas categorias dinamicamente definidas pelo CASE.

Por exemplo, você pode categorizar clientes por faixa etária usando CASE e então usar GROUP BY essa nova categoria para contar quantos clientes estão em cada faixa. Assim como no GROUP BY comum, posições ordinais da coluna CASE podem ser usadas no GROUP BY, mas é preferível usar o alias para clareza.

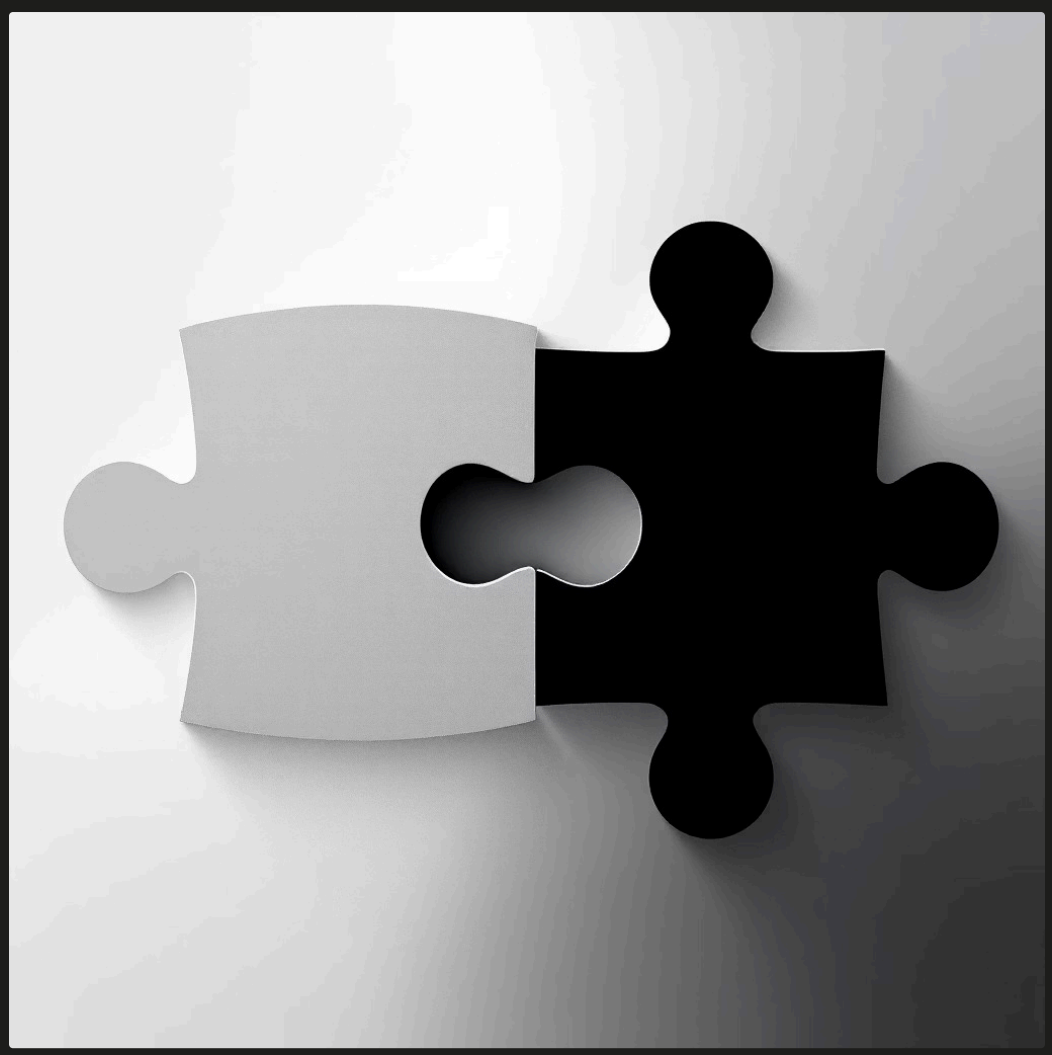
O "Truque Zero/Null" com CASE: Agregações Condicionais

Este é um padrão avançado e altamente eficaz que permite "aplicar filtros diferentes a valores agregados distintos na mesma consulta SELECT".

- **Com** SUM()**:** Quando a condição não é atendida, o CASE deve retornar 0 (ELSE 0 END) para que o valor seja corretamente ignorado na soma sem afetar o total. Exemplo: SUM(CASE WHEN tornado = 1 THEN precipitation ELSE 0 END) somará a precipitação apenas onde houve tornado.
- **Com** MIN() **ou** MAX()**:** Nestes casos, o CASE deve retornar NULL (ELSE NULL END) quando a condição não é atendida, pois funções de agregação como MIN() e MAX() ignoram NULLs. Exemplo: MAX(CASE WHEN tornado = 0 THEN precipitation ELSE NULL END) encontrará a precipitação máxima apenas onde **não** houve tornado.

Este "truque" é inestimável pela sua capacidade de realizar "múltiplas agregações com diferentes critérios" em uma única consulta, evitando a necessidade de várias subconsultas ou JOINs complexos.

6. Unindo Dados com JOINs



Conceitos Fundamentais: O Coração do SQL Relacional

O `JOIN` é amplamente considerado a "funcionalidade que define o SQL" em bancos de dados relacionais. Sua principal função é "associar tabelas" para reunir dados que estão distribuídos logicamente em diferentes entidades.

- **Relacionamento Pai-Filho:** Uma das bases da modelagem relacional é o relacionamento pai-filho. Uma tabela "pai" (ex: `CUSTOMER`) pode estar associada a múltiplos registros de uma tabela "filha" (ex: `CUSTOMER_ORDER`), configurando uma relação "um-para-muitos". O `JOIN` permite navegar por essas relações.

Tipos de JOIN: Escolhendo a Correspondência Certa

- `INNER JOIN`**:** Este é o tipo mais comum e intuitivo. Ele "mescla duas tabelas" e "exclui registros que não têm um valor comum" na condição de união. Ambos os lados devem ter uma correspondência. Sintaxe: `FROM TabelaEsquerda INNER JOIN TabelaDireita ON CondiçãoDeUnião`. Se as colunas tiverem o mesmo nome em ambas as tabelas, é crucial "especificar explicitamente a qual tabela ela pertence" (ex: `CUSTOMER.CUSTOMER_ID`) para evitar ambiguidade.
- `LEFT JOIN`**:** Inclui "todos os registros da tabela 'esquerda'", mesmo que não haja correspondência na tabela "direita". Campos sem correspondência na direita são preenchidos com `NULL`. É extremamente útil para "encontrar registros 'órfãos'" (que não possuem correspondência na tabela direita) ou para garantir que todos os itens de uma lista principal sejam exibidos, mesmo que não tenham dados associados em outra tabela.
- `RIGHT JOIN` **/** `OUTER JOIN` **(**`FULL OUTER JOIN`**):**
  - `RIGHT JOIN`**:** Simétrico ao `LEFT JOIN`, inclui todos os registros da tabela "direita". O SQLite, um banco de dados leve, "não suporta RIGHT JOIN e OUTER JOIN" nativamente, mas seus resultados podem ser replicados com `LEFT JOIN` e inversão das tabelas.
  - `FULL OUTER JOIN`**:** Inclui todos os registros de ambas as tabelas, preenchendo com `NULL` onde não há correspondência.

Associando e Agrupando Múltiplas Tabelas

A flexibilidade do SQL permite "associar múltiplas tabelas" em uma única consulta, simplesmente encadeando operadores `JOIN`. Essa capacidade é fundamental para construir visões complexas de dados.

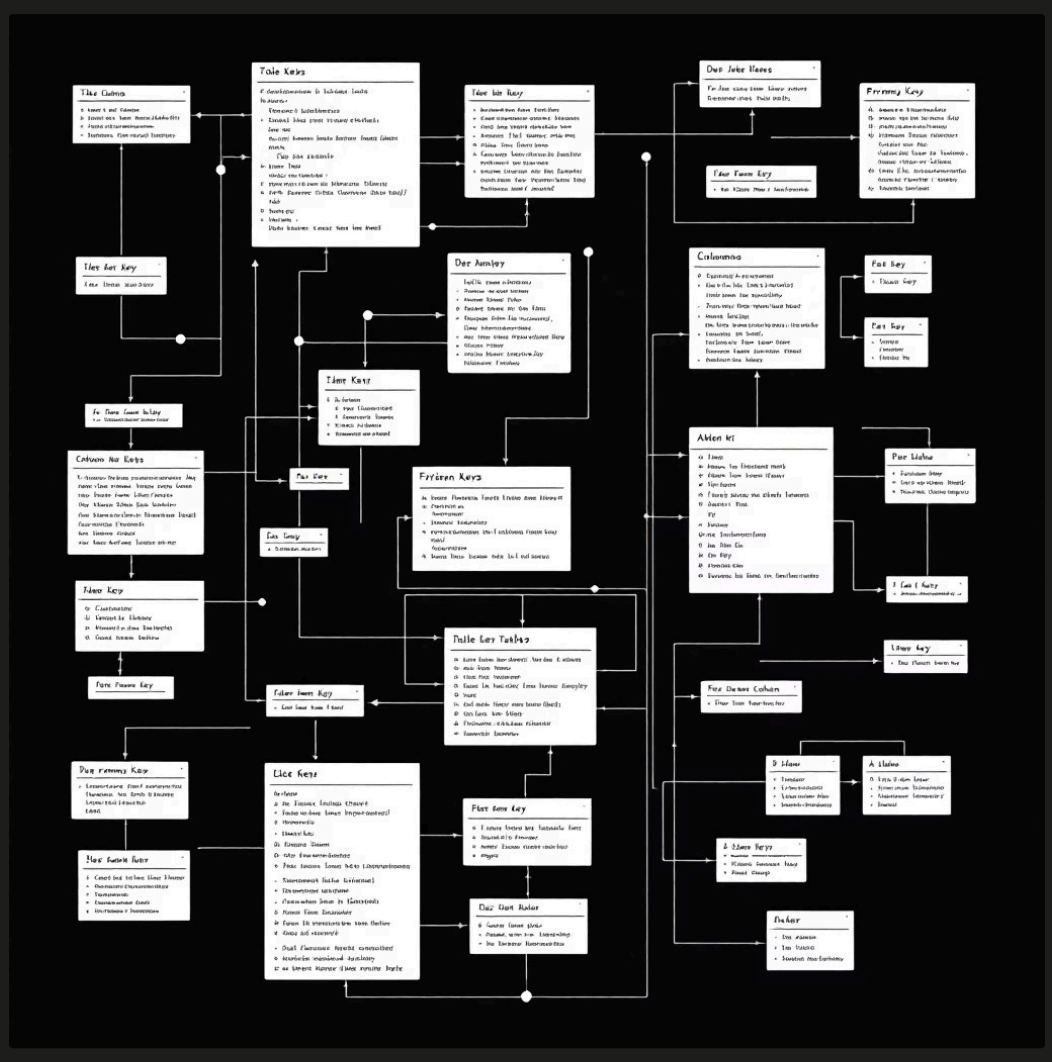
Além disso, os `JOIN`s podem ser perfeitamente combinados com `GROUP BY` e funções de agregação para "resumir dados de várias tabelas". Por exemplo, você pode unir tabelas de clientes, pedidos e produtos para calcular o total de vendas por categoria de produto, por cliente, ou até mesmo o cliente que mais comprou um determinado produto.

- **Uso com** `LEFT JOIN` **e** `coalesce()`**:** Para garantir que todos os registros da tabela da esquerda sejam incluídos, mesmo que não haja correspondência na direita, o `LEFT JOIN` é a escolha certa. Nesses casos, a função `coalesce()` é uma aliada poderosa, pois pode ser usada para converter os `NULL`s resultantes do `LEFT JOIN` em valores padrão (ex: `0`) em colunas agregadas, evitando que nulos afetem negativamente seus cálculos.

```
SELECT
  c.customer_name,
  SUM(COALESCE(o.order_total, 0)) AS total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name
ORDER BY total_spent DESC;
```

ⓘ Dominar os diferentes tipos de `JOIN` e entender quando usar cada um é um marco no domínio do SQL, permitindo que você extraia informações precisas e significativas de qualquer banco de dados relacional.

7. Design de Banco de Dados: Planejamento e Criação de Tabelas



Planejamento do Banco de Dados: A Base do Sucesso

Um "planejamento cuidadoso" é o passo mais crucial na criação de qualquer banco de dados robusto e eficiente. Um design bem pensado evita problemas futuros e garante a integridade dos dados. O design deve ser guiado por:

- **Requisitos do negócio:** Entender as necessidades de dados e as operações que serão realizadas.
- **Normalização:** Decidir como as tabelas serão separadas para evitar redundância e inconsistências.
- **Relacionamentos pai/filho:** Definir as conexões entre as tabelas e como os dados se relacionam.
- **Fornecimento de dados, segurança e administração:** Pensar na forma como os dados serão inseridos, protegidos e mantidos. Ataques como "injeção de SQL" são uma preocupação séria para bancos de dados conectados à web, exigindo medidas de segurança rigorosas desde o projeto inicial.

Chaves Primária e Externa: A Identidade e a Conexão

- **Chave Primária (`PRIMARY KEY`):** É "um campo (ou combinação de campos) que fornece uma identidade exclusiva para cada registro em uma tabela". É fundamental para a "integridade dos dados", impedindo duplicatas e garantindo que cada linha seja única. No SQLite, o tipo `INTEGER PRIMARY KEY` automaticamente atribui IDs sequenciais, simplificando a gestão de chaves.
- **Chave Externa (`FOREIGN KEY`):** É "um campo na tabela-filha que aponta para a chave primária da tabela-pai, estabelecendo um relacionamento". Ao contrário da chave primária, a chave externa não exige exclusividade, pois múltiplos registros filhos podem se relacionar a um único registro pai.

Esquema de Banco de Dados: A Visão Geral

Um "esquema de banco de dados" é um diagrama visual que representa todas as tabelas, suas colunas e, crucialmente, os relacionamentos entre elas (chaves primárias e externas conectadas por setas). É uma ferramenta indispensável para visualizar e verificar a normalização e a estrutura lógica do seu banco de dados.

Criando Tabelas (`CREATE TABLE`)

A instrução `CREATE TABLE` é o comando SQL para declarar uma nova tabela e definir suas colunas.

- Cada coluna possui um **nome**, seu **tipo de dado** (ex: `INTEGER`, `VARCHAR(255)`, `DATE`) e pode ter **restrições**:
  - `NOT NULL`**:** Garante que a coluna não possa ter um valor nulo, exigindo que um dado seja sempre fornecido.
  - `DEFAULT`**:** Define um valor padrão que será automaticamente inserido se nenhum valor for especificado para aquela coluna durante a inserção de um novo registro.

Definindo Chaves Externas: Integridade e Prevenção

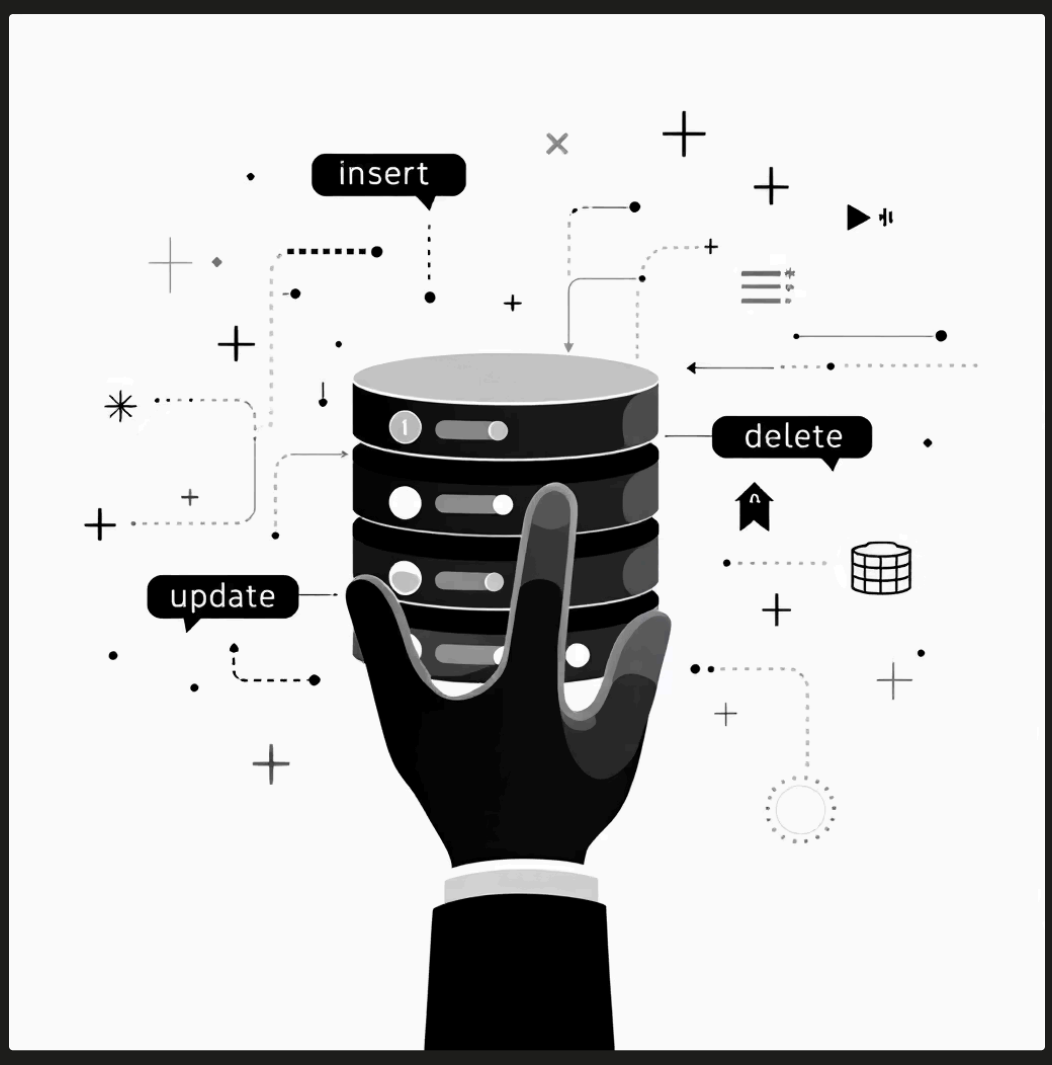
As chaves externas são definidas para "impor a integridade dos dados", garantindo que um valor de chave externa na tabela-filha sempre corresponda a um valor de chave primária existente na tabela-pai. Isso é essencial porque "impede a existência de registros órfãos", ou seja, registros que se referem a algo que não existe (por exemplo, um pedido associado a um cliente que não está no banco de dados).

Criando Views: Tabelas Virtuais para Simplificação

Uma `VIEW` (visualização) é essencialmente uma consulta `SELECT` frequentemente utilizada que é salva no banco de dados com um nome específico.

- Ela se comporta como uma tabela virtual, permitindo que você execute `SELECT`s e `JOIN`s sobre ela como se fosse uma tabela real.
- No entanto, seus dados são sempre derivados da consulta subjacente e, na maioria dos casos, não podem ser modificados diretamente; qualquer alteração nos dados deve ser feita nas tabelas base. Views são ótimas para simplificar consultas complexas, aplicar segurança (exibindo apenas certas colunas) e encapsular lógicas de negócios.

8. Gerenciamento de Dados: Inserir, Excluir e Atualizar



Operações de Gerenciamento de Dados

Este capítulo foca na "manipulação física dos registros" em um banco de dados: "inserir, excluir e atualizar registros". Um "design robusto" de tabelas e relacionamentos é crucial para garantir a integridade dos dados e facilitar a manutenção ao longo do tempo.

Inserindo Registros (INSERT)

A instrução INSERT é usada para "inserir um novo registro" (linha) em uma tabela.

- **Sintaxe básica:** INSERT INTO [NomeDaTabela] (Coluna1, Coluna2) VALUES (Valor1, Valor2). Especifique as colunas e seus respectivos valores.
- Campos com PRIMARY KEY e AUTOINCREMENT geralmente não devem ser preenchidos manualmente, pois o próprio sistema de banco de dados gerencia seus valores.
- Uma inserção falhará se uma coluna definida como NOT NULL não for preenchida e não tiver um valor-padrão associado.

Inserções Múltiplas e Chaves Externas

- **Múltiplas Inserções:** É possível inserir "vários registros de uma só vez" repetindo a cláusula VALUES e separando os conjuntos de valores por vírgula. Esta abordagem é mais eficiente para grandes volumes de dados.
- Também é possível "inserir registros a partir dos resultados de uma consulta SELECT", o que é poderoso para popular tabelas com base em dados existentes.
- **Restrições de Chave Externa:** Chaves externas "impedem a inserção de registros 'órfãos'". Tentativas de inserir um registro filho sem um pai correspondente (ou seja, sem um valor de chave primária válido na tabela pai) resultarão em um erro de integridade referencial.

Excluindo Registros (DELETE, TRUNCATE TABLE, DROP TABLE)

- DELETE**:** Utilizada para "excluir registros" de uma tabela.

Recomendação: Sempre execute um SELECT * FROM [NomeDaTabela] WHERE [Condição]; com a mesma condição WHERE antes de um DELETE para visualizar exatamente quais registros serão afetados e evitar perdas acidentais de dados.

  - DELETE FROM [NomeDaTabela]; exclui **todos** os registros da tabela.
  - DELETE FROM [NomeDaTabela] WHERE [Condição]; exclui registros **condicionalmente**.
- TRUNCATE TABLE**:** Exclui "todos os registros de uma tabela" e reinicia quaisquer contadores de autoincremento. É mais rápido que DELETE sem WHERE para tabelas grandes. No SQLite, essa instrução não é suportada diretamente, mas DELETE FROM [NomeDaTabela]; (sem a cláusula WHERE) oferece otimizações semelhantes.
- DROP TABLE**:** Remove "uma tabela inteira" (estrutura e dados) do banco de dados. É uma instrução perigosa e permanente. Use com extrema cautela.

Atualizando Registros (UPDATE)

A instrução UPDATE é usada para "modificar registros existentes" em uma tabela.

- **Sintaxe básica:** UPDATE [NomeDaTabela] SET Coluna1 = NovoValor1, Coluna2 = NovoValor2.
- As atualizações podem ser aplicadas "condicionalmente com a cláusula WHERE". Sem WHERE, a atualização afetará **todos** os registros da tabela.