

Nesta disciplina, você aprenderá:

- **História e evolução dos bancos de dados** para entender suas tecnologias.
- **Sistemas e arquiteturas de gerenciamento** essenciais para aplicações modernas.
- **Projeto de banco de dados**, da modelagem conceitual à implementação.
- **Diagramas de entidade e relacionamento**, fundamentais para representar dados.
- **Formas normais**, garantindo integridade e eficiência.
- **Uso do PostgreSQL**, incluindo criação, manipulação de tabelas e consultas.

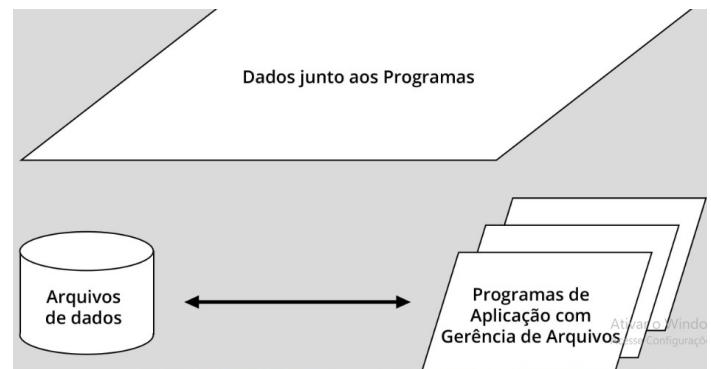
Definição e Evolução dos Bancos de Dados

Um **banco de dados** é uma coleção de dados relacionados, onde dados são fatos registrados com significado. Sempre foi parte essencial dos **sistemas de informação**, que evoluíram com o avanço tecnológico.

Antes dos computadores, os bancos de dados eram registros físicos organizados manualmente. Com a invenção do computador na década de 1940, inicialmente voltado para cálculos matemáticos, percebeu-se seu potencial para **processamento de dados**, graças à arquitetura de John von Neumann.

A revolução ocorreu com a IBM, que introduziu o **disco magnético** (DASD *Direct Access Storage Device*) em 1957, permitindo acesso direto aos dados sem leitura sequencial. Isso deu início à **era do processamento de dados**, na qual sistemas operacionais passaram a gerenciar arquivos armazenados em discos rígidos.

Nessa fase, os programas acessavam dados externos em **arquivos no disco**, substituindo a necessidade de armazenar dados internamente. Esse modelo, baseado em sistemas de arquivos, foi um avanço significativo na computação empresarial e científica, onde linguagens como **COBOL** e **Fortran** eram amplamente utilizadas.



Evolução dos Sistemas de Arquivos para Bancos de Dados

O modelo de **processamento de dados com sistema de arquivos** foi amplamente usado desde a adoção do disco magnético e ainda persiste em sistemas legados. Um exemplo disso foi a alta demanda por programadores **COBOL** durante a pandemia de COVID-19 para manter sistemas governamentais nos EUA.

Com o tempo, a necessidade de evitar a **repetição de código** nos programas levou ao surgimento dos **bancos de dados**. Nos sistemas antigos, cada aplicação precisava conter um módulo próprio para gerenciar arquivos de dados. Isso dificultava a manutenção, pois diferentes departamentos usavam os mesmos dados, exigindo cópias redundantes do código.

A solução surgiu nos anos 1960 com os **Sistemas de Banco de Dados (SBD)**, que retiraram dos programas essa responsabilidade. A gestão dos dados passou a ser feita por um software

intermediário: o **Sistema de Gerenciamento de Banco de Dados (SGBD)**, centralizando o armazenamento e acesso aos dados.

Essa propriedade dos sistemas de banco de dados é denominada de **independência entre dados e programas**, uma diferença primordial em relação aos sistemas de arquivos.



Modularização e Evolução dos Sistemas de Informação

A modularização dos sistemas de informação distribuiu responsabilidades entre os **programas de aplicação** e o **SGBD**. Os programas focam nas funcionalidades da aplicação, enquanto o **SGBD** gerencia o acesso e a manipulação dos dados, funcionando como um serviço de **back-end**.

É importante distinguir o **Sistema de Banco de Dados (SBD)**, que inclui o **SGBD**, os **programas de aplicação** e os próprios **bancos de dados**.

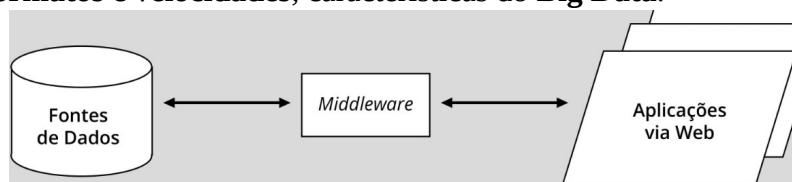
Com os avanços tecnológicos, surge a questão: **qual modelo é mais eficiente** para lidar com o mesmo volume de dados? Essa comparação entre **modelo monolítico (dados junto dos programas), sistema de arquivos e banco de dados** será abordada ao final do módulo.

Sistemas de Informação na Web

A revolução da **World Wide Web** transformou os sistemas de informação. Com novas linguagens e formas de armazenamento, o **SGBD** se tornou parte de um **middleware**, intermediando aplicações e **múltiplas fontes de dados**.

Os sistemas modernos não se limitam a **dados estruturados** de bancos tradicionais, mas lidam com **grandes volumes de dados em diversos formatos e velocidades**, características do **Big Data**.

Além disso, aplicações Web são desenvolvidas em diferentes plataformas, todas conectadas à internet e integradas à **computação em nuvem (Cloud Computing)**.



Evolução dos Sistemas de Banco de Dados

Na década de 1960, dois SGBDs foram desenvolvidos de forma independente:

- **IDS** (Criado por Charles Bachman) – Utilizava uma estrutura de **redes (grafos)**, originando os **network databases**.
- **IMS** (Criado pela IBM) – Baseado em **árvores (hierarquias)**, dando origem aos **hierarchical databases**.

Ambos eram **navegacionais**, pois os dados eram acessados sequencialmente, registro por registro. Outros SGBDs seguiram essa abordagem, como **DMS** e **IDMS**.

Apesar dos avanços tecnológicos, muitos **sistemas legados (sistemas de software antigos, mas ainda em uso dentro de uma organização)** ainda utilizam esses bancos, evidenciado pela alta demanda por **programadores COBOL** durante a pandemia de COVID-19.

O Modelo Relacional de Banco de Dados

A grande revolução nos bancos de dados ocorreu entre as décadas de 1960 e 1970, quando **Edgar F. Codd**, pesquisador da IBM, publicou o artigo *A Relational Model of Data for Large Shared Data Banks*. Esse trabalho introduziu o **modelo relacional**, que organiza **dados em tabelas (relações)**, substituindo a estrutura de grafos dos bancos navegacionais.

Codd desenvolveu a **Álgebra Relacional** e o **Cálculo Relacional**, formando a base teórica do modelo. Sua simplicidade e fundamentação matemática impulsionaram implementações tanto na IBM quanto no meio acadêmico, especialmente na **Universidade da Califórnia, Berkeley (UCB)**.

Principais SGBDs Relacionais

- **IBM**: Desenvolveu o **System R**, que originou o **SQL/DS**, posteriormente renomeado **DB2**, um dos líderes no mercado.
- **Oracle**: Criado pela empresa que evoluiu de SDL para RSI e, finalmente, **Oracle Corporation**, tornou-se o maior SGBD relacional comercial.
- **MySQL**: Após a aquisição da Sun Microsystems pela Oracle em 2010, tornou-se um dos principais SGBDs para aplicações Web, formando a pilha **LAMP** (Linux, Apache, MySQL, PHP).
- **Ingres**: Projeto acadêmico da UCB, que originou tanto uma versão comercial (**Ingres DBMS**) quanto o **Sybase**, que posteriormente deu origem ao **SQL Server** da Microsoft.
- **PostgreSQL**: Evolução do projeto **Postgres (Post-Ingres)**, incorporou conceitos de programação orientada a objetos e SQL, tornando-se o **SGBD open source mais avançado do mundo**.

PostgreSQL e **MySQL** são amplamente utilizados no ensino de bancos de dados relacionais devido à popularidade e licença livre.

Outros Modelos de SGBDs

O ranking do **DB-Engines** destaca como líderes do modelo relacional **Oracle**, **MySQL**, **Microsoft SQL Server**, **PostgreSQL** e **IBM DB2**. No entanto, esses SGBDs são classificados como **multimodelo**, pois incorporam funcionalidades além do relacional:

- **Oracle**: Relacional e multimodelo (documentos, grafos e RDF).
- **MySQL**: Relacional e multimodelo (documentos).
- **SQL Server**: Relacional e multimodelo (documentos e grafos).
- **PostgreSQL**: Relacional e multimodelo (documentos).
- **IBM DB2**: Relacional e multimodelo (documentos e RDF).

Modelos Não Relacionais (NoSQL)

Embora o **modelo relacional** seja predominante em sistemas empresariais devido à sua robustez e padronização com SQL, algumas aplicações exigem estruturas mais flexíveis, como **Big Data**, **IoT** e **Data Science**. Isso impulsionou o crescimento dos **bancos de dados NoSQL**, que não seguem o modelo relacional e, em muitos casos, não utilizam SQL.

O site **DB-Engines Ranking** lista diversos **modelos de banco de dados NoSQL**, incluindo:

- **Chave-Valor:** Redis, Amazon DynamoDB.
- **Documentos:** MongoDB, Microsoft Azure CosmosDB.
- **Séries temporais:** InfluxDB, Prometheus.
- **Grafos:** Neo4J, ArangoDB.
- **Orientado a objetos:** InterSystems Caché, ObjectStore.
- **Motores de busca:** Elasticsearch, Splunk.
- **RDF:** Marklogic, Virtuoso.
- **Colunar:** Cassandra, HBase.
- **Eventos:** Event Store, IBM DB2 Event Store.

Modelo Navegacional

O **modelo navegacional**, adotado nos primeiros SGBDs das décadas de 1960 e 1970, ainda é utilizado em sistemas legados. Exemplos incluem **IMS** e **IDMS**, que foram predecessores do modelo relacional.

Registros são armazenados em estruturas hierárquicas ou em redes interligadas. Os dados são organizados em uma estrutura de árvore (modelo hierárquico) ou em grafos interligados (modelo de rede). O acesso aos registros é feito por meio de ponteiros que indicam conexões diretas entre os dados.

Sistemas de arquivos vs bancos de dados

Característica	Sistema de Arquivos 	Banco de Dados 
Estrutura	Hierárquica (pastas e arquivos)	Relacional (tabelas) ou NoSQL (documentos, grafos, colunas)
Acesso aos Dados	Manual (arquivos abertos e lidos por programas)	Consultas estruturadas via SQL ou APIs
Redundância	Alta (dados podem ser repetidos em vários arquivos)	Baixa (normalização elimina duplicações)
Consistência	Nenhum mecanismo interno de consistência	Regras e restrições garantem integridade
Segurança	Controle básico (permissões de pastas)	Controle avançado (criptografia, usuários, permissões)
Velocidade de Busca	Lenta para grandes volumes de dados	Otimizada com índices e consultas
Manipulação de Dados	Processamento manual	Operações complexas e transações ACID

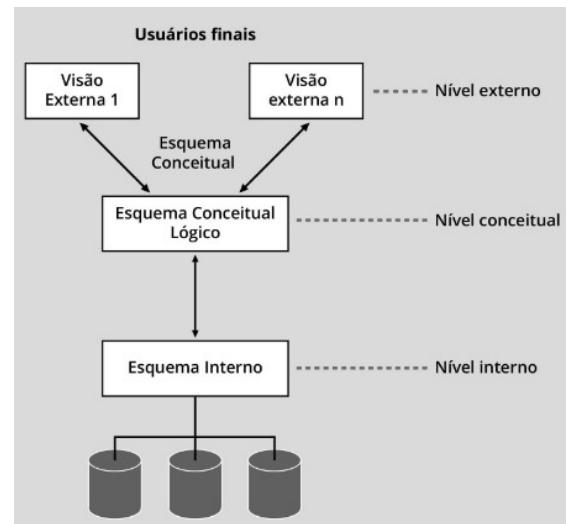
Diferenças entre Sistema de Arquivos e Sistema de Banco de Dados

A principal diferença entre sistemas de arquivos e bancos de dados é a **independência dos dados em relação aos programas**. Essa independência decorre da **arquitetura de três esquemas**, conforme o modelo **ANSI/SPARC**, que separa as aplicações do armazenamento físico:

- **Nível Externo:** Contém as diferentes visões dos usuários.
- **Nível Conceitual:** Descreve a estrutura lógica do banco de dados.
- **Nível Interno:** Representa o armazenamento físico dos dados.

Com base nessa arquitetura, existem **dois tipos de independência de dados:**

- **Independência lógica:** Permite alterações no esquema conceitual sem afetar as visões externas dos usuários.
- **Independência física:** Permite mudanças na organização física dos dados sem impactar a estrutura lógica.



Outras Características dos Bancos de Dados

- **Natureza autocontida:** Além dos dados, um banco de dados armazena metadados (descrição da estrutura e restrições dos dados) no catálogo do sistema.
- **Abstração de dados:** Modelos de dados ocultam detalhes de armazenamento, permitindo múltiplas visões lógicas.
- **Compartilhamento de dados e processamento concorrente:** Suporte a múltiplos acessos simultâneos ao banco de dados.

Modelos de Dados

Os modelos de dados que garantem a **abstração e independência lógica e física** são classificados em **físicos, lógicos e conceituais**.

Modelos Físicos

- Definem como os dados são armazenados no computador, detalhando tipos de arquivos, formatos, ordenação de registros e caminhos de acesso. Dependem do SGBD e do sistema operacional utilizado.

Modelos Lógicos

- Representam a implementação do banco de dados de forma abstrata, sem expor detalhes físicos. O principal é o **modelo relacional(SQL)**, baseado em tabelas (relações matemáticas), criado por **Edgar Codd** e amplamente usado em sistemas organizacionais. Além disso, há **modelos não relacionais (NoSQL)**, que seguem diferentes abordagens.

Modelos Conceituais

- Expressam a visão dos dados sob a perspectiva do usuário final. O **Modelo Entidade-Relacionamento (ER)**, criado por **Peter Chen** em 1976, auxilia na compreensão do modelo relacional e é usado na **modelagem conceitual de dados**.

A modelagem ER foi incorporada na **UML (Unified Modeling Language)**, que padroniza a modelagem orientada a objetos. O **modelo de classes da UML**, onde **entidades** são representadas por **classes** e **relacionamentos** por **associações**, substitui frequentemente o modelo ER no desenvolvimento de sistemas.

Definição Atualizada de Banco de Dados

Banco de dados é uma coleção autodescritiva de dados relacionados, com significado lógico inerente, acessível concurrentemente por múltiplos usuários com diferentes visões.

Vantagens e Desvantagens da Abordagem de Banco de Dados

A abordagem de Sistemas de Banco de Dados (SBD) oferece funcionalidades que superam sistemas baseados em arquivos, como controle de redundância, compartilhamento seguro e integridade dos dados.

Principais Vantagens

- **Controle da Redundância:** Reduz inconsistências, duplicação de esforços e desperdício de armazenamento. O SGBD equilibra a centralização dos dados com redundância estratégica para otimizar consultas.
- **Compartilhamento e Concorrência:** Múltiplos usuários acessam dados simultaneamente, com controle para garantir atomicidade, consistência, isolamento e durabilidade (ACID) das transações.
- **Controle de Acesso:** Implementa segurança via senhas, restrições de acesso, permissões específicas e limitação de software administrativo.
- **Múltiplas Interfaces:** Suporte a diferentes perfis de usuários, incluindo linguagens de consulta, interfaces gráficas, menus e comandos em linguagem natural.
- **Relacionamentos Entre Dados:** Representação dos vínculos entre informações conforme o modelo lógico adotado.
- **Cumprimento de Restrições de Integridade:** Garante regras como tipos de dados, domínios, unicidade, integridade referencial e outras restrições semânticas. Pode delegar esse controle ao SGBD ou aos programas de aplicação.
- **Backup e Recuperação:** Mecanismos de backup e restore garantem a integridade do banco de dados após falhas.
- **Suporte a Múltiplos Usuários Simultâneos:** O SGBD mantém a concorrência e integridade das transações, relaxando as regras ACID conforme necessário para equilibrar desempenho e segurança.

Propriedades ACID

- **Atomicidade:** A transação é executada completamente ou falha completamente. Se ocorrer uma falha, o sistema reverte as mudanças.
- **Consistência:** A transação leva o banco de dados de um estado válido a outro, respeitando as regras de integridade.
- **Isolamento:** Transações simultâneas não interferem umas nas outras, garantindo que o resultado seja o mesmo como se fossem executadas sequencialmente.
- **Durabilidade:** Após a confirmação da transação, seus resultados permanecem válidos, mesmo em caso de falhas.

Vantagens do SBD:

- Estabelecimento de padrões de uso de dados na organização.
- Redução do tempo de desenvolvimento de aplicações.
- Flexibilidade na manutenção dos dados.
- Disponibilidade de dados atualizados em toda a organização.
- Economia de escala.

Desvantagens do SBD:

- Sobrecarga de desempenho do sistema devido à presença do SGBD.
- Custo e esforço adicional para capacitação e oferta de funcionalidades sofisticadas.

Casos onde é preferível não usar SBD:

- Aplicações simples com dados estáticos e bem definidos, que não se espera alterar (ex.: censo demográfico)
- Aplicações de tempo real com requisitos rígidos que não suportam o overhead do SGBD (ex.: controle de tráfego aéreo)
- Sistemas embarcados com poucos dados e requisitos estritos de tempo real (ex.: piloto automático)
- Sistemas monousuários de uso sem concorrência, típicos de desktops (ex.: prontuário eletrônico de um único consultório médico)

Exceções:

- Aplicações que requerem grandes volumes de dados com dinâmica incompatível com SGBDs tradicionais, como a IoT, podem ser atendidas por bancos de dados NoSQL.

***Overhead** em SGBD refere-se ao consumo adicional de recursos (CPU, memória, disco, tempo de processamento) necessário para gerenciar e manter a integridade, segurança e eficiência do banco de dados.

Papéis em sistemas de bancos de dados

Usuários finais: Beneficiários dos sistemas de computação, com ou sem uso de SGBD.

Administrador de aplicações: Função técnico-gerencial que mantém sistemas de aplicação e oferece suporte aos usuários. Exemplo: administrador de **ERP (Enterprise Resource Planning)**.

Administrador de desenvolvimento: Função técnica que pode ser dividida em equipes dependendo da complexidade do sistema, usando ferramentas do ambiente de desenvolvimento.

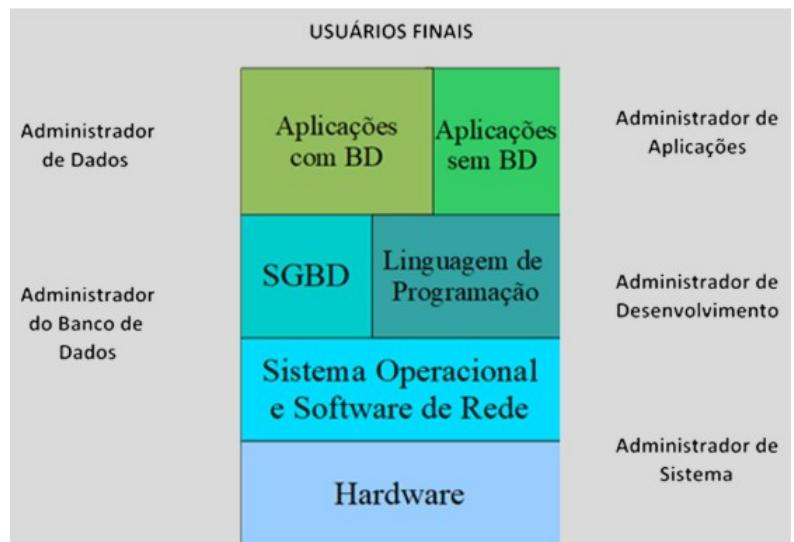
Administrador de dados: Função gerencial que define políticas, responsabilidades, regras de negócio e padrões de dados na organização.

Administrador do banco de dados:

Função técnica responsável pela criação e manutenção dos bancos de dados no SGBD, suportando as equipes de desenvolvimento.

Administrador de sistema: Mantém o sistema de computação em funcionamento, focando no hardware, sistema operacional e interfaces com outros softwares.

Observação: A distribuição dos papéis administrativos varia conforme o porte da organização e do sistema de computação, podendo ser exercidos por uma ou mais pessoas.



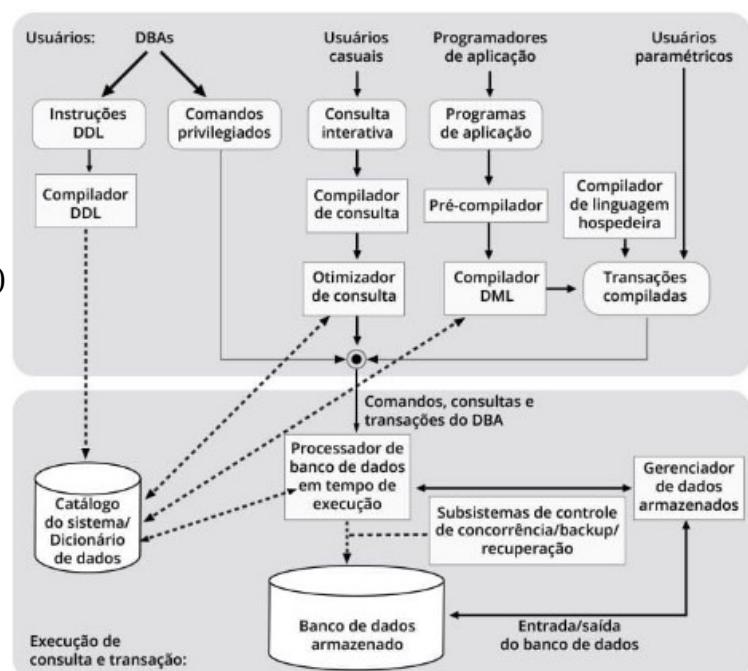
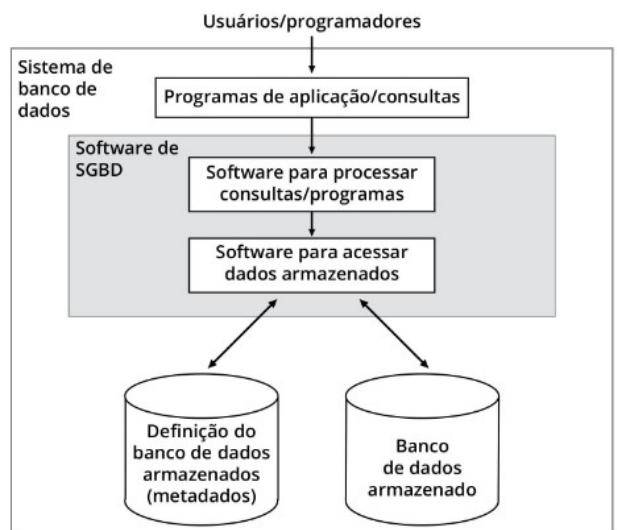
Metadados são a **estrutura do banco de dados**, definida conforme o modelo lógico de implementação. Modelar um banco segundo esse modelo significa esquematizá-lo de acordo com seus construtores. No modelo relacional, por exemplo, o esquema é composto por tabelas e colunas. Comandos de definição de dados (criação, alteração ou remoção de tabelas) modificam esse esquema.

O estado ou instância do banco de dados refere-se ao conteúdo armazenado em um momento específico. Qualquer operação de inserção, atualização ou remoção altera esse estado, gerando uma nova instância do banco.

O termo "banco de dados" vem do inglês "database", registrado pela primeira vez em 1962. O termo "data bank" é menos comum, aparecendo, por exemplo, no artigo de Edgar Codd sobre o modelo relacional (1970).

Em outros idiomas, o equivalente é **base de dados** (espanhol), **base de données** (francês), **Datenbank** (alemão) e **banca dati** (italiano). Em português, "banco de dados" refere-se ao ambiente geral do sistema, enquanto "base de dados" se refere ao conteúdo armazenado. Por exemplo, a "base de dados" da Receita Federal representa os dados dos contribuintes, não o sistema que os gerencia.

Um Sistema de Gerência de Banco de Dados (SGBD) é composto por diversos módulos, divididos entre o processamento de consultas e aplicações (parte superior) e o acesso a metadados e dados armazenados (parte inferior).



Usuários do SGBD

- **Administrador de Banco de Dados (ABD/DBA):** Utiliza a **Linguagem de Definição de Dados (LDD/DDL)** para criar, alterar e remover objetos no banco de dados (CREATE, ALTER, DROP). Também gerencia permissões de acesso com a **Linguagem de Controle de Dados (LCD/DCL)**, usando comandos como GRANT e REVOKE. Suas operações passam pelo **processador de runtime** do SGBD.
- **Usuários casuais:** Realizam consultas interativas ad hoc via interface, geralmente com comandos SELECT. As consultas são processadas por um **compilador de linguagem de consulta** e otimizadas antes de serem enviadas ao núcleo do SGBD.
- **Programadores de aplicações:** Desenvolvem programas em linguagens como Java, PHP ou Python, incluindo comandos de manipulação de dados (LMD/DML) como INSERT, UPDATE e DELETE, que diferem dos comandos LDD, que manipulam metadados.
- **Usuários paramétricos:** Interagem com o sistema fornecendo parâmetros específicos para execução de transações, como no caso de reservas de passagens aéreas.

Processamento das Aplicações

- **Programas de aplicação:** Misturam comandos da linguagem hospedeira com consultas e manipulações de dados.
- **Pré-compilador:** Separa os comandos para os compiladores apropriados.
- **Compiladores:** Geram o código das aplicações na forma de transações executáveis.
- **Transações compiladas:** São enviadas ao processador de runtime para execução.

O **processador de runtime** (ou **runtime processor**) em um banco de dados é o componente responsável por **executar as consultas e comandos enviados ao SGBD em tempo de execução**. Ele interpreta, optimiza e executa as instruções, garantindo que os dados sejam manipulados corretamente.

Acesso aos Dados e Processamento

O **processador de runtime** é o núcleo do SGBD, determinando sua eficiência e funcionalidades. Ele acessa tanto os dados armazenados quanto o catálogo do sistema para garantir a consistência dos objetos do banco. O acesso à base de dados é gerenciado por subsistemas de controle de concorrência, backup e recuperação. O gerenciador de dados armazenados gerencia operações de entrada e saída.

SGBDs evoluíram desde os anos 1960, tornando-se indispensáveis para organizações. Atualmente, existem mais de 300 SGBDs no mercado, sendo cerca de 130 do modelo relacional, conforme o ranking do DB-Engines. A Wikipédia oferece uma comparação detalhada de SGBDs relacionais na página "Comparison of relational database management systems".

PostgreSQL: Um Exemplo de SGBD Relacional

O PostgreSQL surgiu do projeto Ingres, da Universidade da Califórnia em Berkeley, nos anos 1970. Inicialmente chamado **Postgres (Post Ingres)** e depois **Postgres 95**, recebeu o nome definitivo após a adoção da linguagem SQL.

Originalmente, Ingres e Postgres utilizavam a linguagem **QueL (Query Language)**, concorrente da SQL da IBM. Com o reconhecimento da SQL como padrão pelo **ANSI** e **ISO**, o Postgres adotou a linguagem, tornando-se o **PostgreSQL**.

Reconhecido como "*o banco de dados relacional open source mais avançado do mundo*", o PostgreSQL tem documentação de referência global e é amplamente utilizado no ensino e em grandes empresas.

O PostgreSQL é um **SGBD relacional-objeto**, baseado em tabelas, mas com extensões de orientação a objetos. Ele suporta grande parte do **padrão SQL** e inclui funcionalidades como consultas complexas, **triggers**, visões materializadas atualizáveis e controle de concorrência multiversionado. Além disso, pode ser estendido com novos tipos de dados, funções, operadores, métodos de indexação e linguagens procedurais.

Modelo Cliente/Servidor do PostgreSQL

O **PostgreSQL** adota um modelo **cliente/servidor**, comum em SGBDs relacionais empresariais. Uma sessão envolve:

- **Back end:** Processo servidor (*postgres*) que gerencia arquivos do banco, aceita conexões de clientes e executa comandos em nome deles.
- **Front end:** Aplicações clientes que interagem com o banco, como **ferramentas de linha de comando (psql)**, **interfaces gráficas (pgAdmin)**, servidores web e sistemas de manutenção.

Características e Suporte

O PostgreSQL roda nos principais sistemas operacionais, incluindo **Linux, UNIX e Windows**. Suporta **transações ACID** (atomicidade, consistência, isolamento e durabilidade), além de **chaves estrangeiras, junções, visões, funções, triggers e procedimentos armazenados** em diversas linguagens, como **Java, Perl, Python, Ruby, Tcl, C/C++ e PL/pgSQL** (similar ao PL/SQL do Oracle).

Recursos Avançados

Como banco de dados **empresarial**, inclui:

- **Controle de concorrência multiversão (MVCC).**
- **Recuperação Point-in-Time (PiTR).**
 - Permite restaurar um banco de dados a um estado exato em um momento específico no passado. Isso é feito usando backups base (full) + logs de transação (WAL – Write-Ahead Logging) para reverter mudanças até o ponto desejado.
- **Tablespaces e replicação assíncrona.**
- **Transações aninhadas (savepoints).**
- **Backups online/hot.**
- **Planejador/otimizador de consultas avançado.**
- **Write Ahead Logging (WAL) para tolerância a falhas.**
 - Mecanismo de registro no PostgreSQL que garante a durabilidade (ACID), escrevendo alterações primeiro em um log antes de aplicá-las no banco de dados.

Log: registro sequencial de eventos, operações ou transações ocorridas em um sistema, utilizado para **monitoramento, diagnóstico e recuperação**.

Escalabilidade e Metadados

O PostgreSQL é **altamente escalável**, suportando **centenas de usuários simultâneos e terabytes de dados**. Seu **catálogo de sistema relacional** permite múltiplos schemas por banco e pode ser acessado via **Information Schema**, conforme o padrão SQL.

- **Information Schema** é uma **view padrão** em sistemas de gerenciamento de banco de dados (SGBDs) que fornece informações sobre os **objetos e estruturas** do banco de dados, como tabelas, colunas, índices, e permissões. Ele oferece uma maneira **consistente e padronizada** de acessar metadados, independente do SGBD utilizado.

Recursos Avançados do PostgreSQL

O PostgreSQL suporta **índices compostos, parciais e funcionais**, utilizando métodos como **B-tree, R-tree, hash e GiST**. O **GiST** é um framework avançado usado em projetos como:

- **OpenFTS** – Motor de busca full-text com indexação online e ranking de relevância.
- **PostGIS** – Extensão para dados geográficos, permitindo o uso do PostgreSQL como banco de dados espacial para **Sistemas de Informação Geográfica (SIG)**.

Funcionalidades Adicionais

- **Herança de tabelas:** Permite derivar novas tabelas de outras, similar à herança em POO, suportando herança simples e múltipla.
- **Sistema de regras:** Também chamado de reescrita de consultas, transforma dinamicamente operações em alternativas (**cláusula INSTEAD OF**).
- **Sistema de eventos:** Comunicação entre processos via *LISTEN* e *NOTIFY*, permitindo monitoramento em tempo real de eventos no banco.

Licença e Uso Comercial

O código do PostgreSQL é **open source**, sob licença **BSD**, permitindo uso, modificação e distribuição livremente, seja como código aberto ou fechado. Isso possibilita não apenas a utilização empresarial, mas também o desenvolvimento de produtos próprios baseados na tecnologia.

Desde 2004, a **EnterpriseDB** oferece uma versão empresarial do PostgreSQL com extensões adicionais para o mercado corporativo.

Introdução à Construção de um Banco de Dados

A criação de um banco de dados envolve a **automatização de um negócio** ou parte dele. Segundo **Elmasri e Navathe (2019)**, um banco de dados representa um **aspecto do mundo real**, chamado de **minimundo** ou **universo de discurso**, tornando essencial o conhecimento do funcionamento do negócio.

O processo de construção é dividido em **fases bem definidas**, utilizando **modelos de dados** para facilitar a compreensão da estrutura do banco, sem detalhamento da representação física dos dados.

Na etapa de **projeto conceitual**, será construído um **Diagrama de Entidade e Relacionamento (DER)**, baseado em **entidades** e **relacionamentos**. Esse diagrama passa por revisões para incorporar novos requisitos, sendo um **processo incremental**.

Por fim, será abordada a **modelagem de atributos**, concluindo o ciclo de aprendizagem.

Levantamento de Requisitos e Projeto Conceitual

O desenvolvimento de sistemas envolve **duas fases essenciais**: **levantamento de requisitos** e **projeto conceitual** de bancos de dados.

O **projeto de um banco de dados** segue quatro etapas principais: **levantamento de requisitos, projeto conceitual, projeto lógico e projeto físico**. Para isso, é necessário entender as **rotinas do negócio** e identificar as necessidades de gestão de dados.

Levantamento de Requisitos

Nessa etapa, o profissional entrevista usuários para compreender o funcionamento do negócio e documentar **requisitos de dados** de forma detalhada.

Exemplo:

Ao projetar um banco de dados para uma escola de treinamentos em TI, os requisitos identificados incluem:

- **Cursos**: código único, nome, descrição e carga horária.
- **Clientes**: código único, nome, data de nascimento, CPF, e-mail e telefone.
- **Inscrição**: data da inscrição e, se houver cancelamento, data do evento. **Um cliente pode se inscrever em vários cursos.**

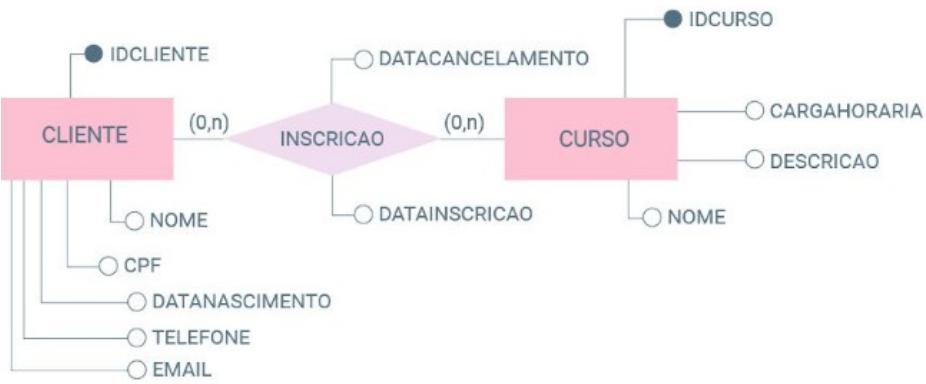
Projeto Conceitual

Com os requisitos de dados definidos, inicia-se o **projeto conceitual**, onde se constrói um **modelo de dados de alto nível**. Essa fase formaliza os requisitos mapeados, sem considerar detalhes de armazenamento.

O principal instrumento do projeto conceitual é o **Diagrama de Entidade e Relacionamento (DER)**, que representa visualmente os objetos e seus relacionamentos. Seus três elementos essenciais são:

- **Entidades**: representadas por **retângulos** com seus respectivos nomes.
- **Relacionamentos**: ilustrados por **losangos** conectando as entidades envolvidas.
- **Atributos**: ligados graficamente às entidades ou relacionamentos correspondentes.

A seguir, um exemplo de **DER baseado nos requisitos de dados levantados**.



DER construído a partir dos requisitos de dados da Escola

Esquema Conceitual e Notação UML

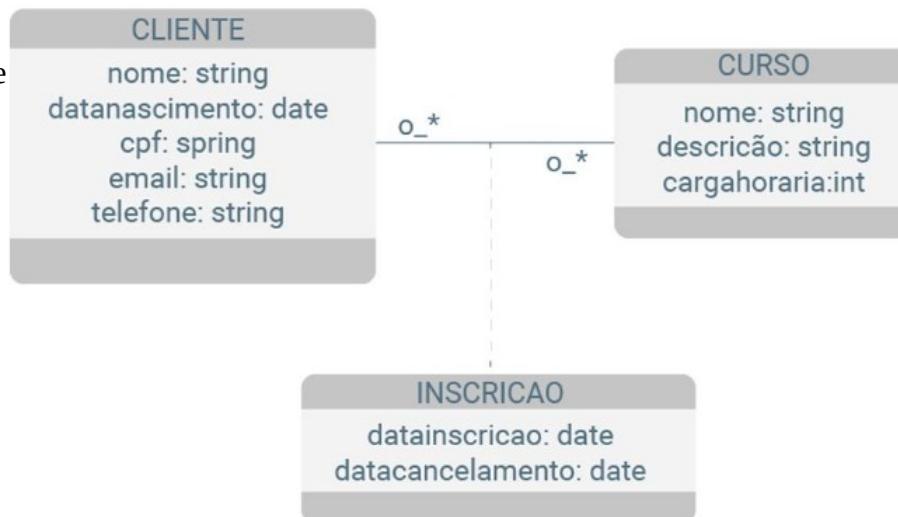
O **Diagrama de Entidade e Relacionamento (DER)** mostra duas entidades essenciais: **CLIENTE** e **CURSO**, relacionadas por inscrições de clientes em um ou mais cursos. O DER é uma ferramenta para garantir que todos os requisitos de dados sejam atendidos, sem conflitos, e fornece uma representação precisa do funcionamento do negócio.

Não há uma notação-padrão para o DER; a escolha depende da preferência do profissional ou da empresa. Em muitos projetos de software, a **UML** é utilizada como uma alternativa para o DER, principalmente com o **diagrama de classes**. Este diagrama é composto por três seções:

1. **Superior**: Nome da classe.
2. **Central**: Atributos e seus tipos de dados.
3. **Inferior**: Operações associadas à classe.

As **associações** entre classes são representadas por linhas, e atributos dos relacionamentos ficam em caixas conectadas à associação.

Após o DER e levantamento de requisitos, a próxima fase será o **projeto lógico**, onde o modelo de dados será detalhado conforme o SGBD escolhido.



Esquema conceitual Escola na notação do diagrama de classes UML

Projeto Lógico

O **projeto lógico** transforma o modelo conceitual em um modelo lógico, dependendo do tipo de SGBD escolhido. Os principais modelos lógicos incluem:

- Rede
- Hierárquico
- Relacional
- Orientado a objeto
- Grafos
- Chave-valor
- XML

O modelo **relacional** é o mais utilizado, com exemplos como Oracle, MySQL, PostgreSQL, SQLite e SQL Server.

O **modelo relacional** surgiu nos anos 1970 e organiza os dados em **tabelas** com nomes e colunas. A conversão do **DER** (Diagrama de Entidade e Relacionamento) para o modelo relacional é feita com regras bem definidas, convertendo as entidades em tabelas (por exemplo, **CLIENTE** e **CURSO**) e representando relacionamentos como tabelas também (exemplo: **INSCRIÇÃO**). No projeto lógico,



Tabelas originadas do DER da Escola

ainda não são definidos os detalhes dos atributos, como tipo de dados e tamanho, apenas a vinculação às tabelas.

Além da representação visual, as tabelas podem ser descritas textualmente.

Tabelas Originadas do DER

CLIENTE

- Atributos: idcliente, nome, datanascimento, CPF, email, telefone

CURSO

- Atributos: idcurso, nome, cargahoraria, descricao

INSCRIÇÃO

- Atributos: idcurso, idcliente, datainscricao, datacancelamento

Observações:

- CLIENTE** é caracterizado por um identificador e atributos como nome, data de nascimento, CPF, e-mail e telefone.
- CURSO** tem um identificador e atributos como nome, carga horária e descrição.
- INSCRIÇÃO** associa um cliente a um curso, com atributos como data de inscrição e data de cancelamento.

Com isso, concluímos as fases do projeto de banco de dados e seguimos para a construção do projeto físico.

No **projeto físico**, definimos os detalhes de implementação dos objetos do banco de dados, como tipos de dados e tamanho das colunas das tabelas, além de especificar se são obrigatórias ou opcionais. Os relacionamentos são representados por **chaves estrangeiras**. Esse processo é geralmente facilitado por ferramentas gráficas de modelagem, muitas vezes disponíveis online com recursos limitados.

Para o exemplo, foi escolhida a ferramenta **Vertabelo**, que é funcional e gratuita para fins educacionais. O SGBD escolhido para a modelagem foi o **PostgreSQL**, com o modelo físico adaptado para esse sistema.



No **projeto físico**, os detalhes de implementação dos objetos do banco de dados são definidos, como o tipo de dados, o tamanho das colunas e as restrições (PK, FK, N). Diferente do modelo lógico, no projeto físico cada coluna de tabela é especificada com esses detalhes. Para a criação do esquema do banco, usamos a **Linguagem de Consulta Estruturada SQL**, especificamente a **Linguagem de Definição de Dados (DDL)**. Um script DDL é utilizado para criar as tabelas do banco de dados, como demonstrado no exemplo, onde cada tabela começa com o comando **CREATE TABLE**.

No exemplo de um banco de dados de escola, cada tabela possui um identificador único: **idcliente** para clientes e **idcurso** para cursos. A tabela **INSCRIÇÃO** usa um identificador composto pelas colunas **idcurso** e **idcliente**. As linhas 21 e 22 do script garantem que a inscrição envolva necessariamente um cliente e um curso previamente cadastrados no banco de dados, assegurando a integridade dos dados.

Ao longo do estudo, vimos as fases de um **projeto de banco de dados**, que incluem: levantamento de requisitos de dados, construção do modelo de **Entidade e Relacionamento (DER)**, desenvolvimento do modelo lógico e, finalmente, a implementação do modelo físico. Esse processo não é estático, pois os requisitos podem evoluir com o tempo, seja por mudanças nas necessidades do negócio, seja por adaptações legais ou para otimizar a eficiência no uso do banco de dados.

```

1 CREATE TABLE CLIENTE (
2   idcliente int NOT NULL,
3   nome char(90) NOT NULL,
4   datanascimento date NOT NULL,
5   CPF char(12) NOT NULL,
6   email char(50) NOT NULL,
7   telefone char(12) NULL,
8   PRIMARY KEY (idcliente) );
9 CREATE TABLE CURSO (
10  idcurso int NOT NULL,
11  nome char(90) NOT NULL,
12  cargahoraria int NOT NULL,
13  descricao char(120) NOT NULL,
14  PRIMARY KEY (idcurso) );
15 CREATE TABLE INSCRICAO (
16  idcurso int NOT NULL,
17  idcliente int NOT NULL,
18  datainscricao date NOT NULL,
19  datacancelamento date NULL,
20  PRIMARY KEY (idcurso,idcliente),
21  FOREIGN KEY (idcliente) REFERENCES CLIENTE (idcliente),
22  FOREIGN KEY (idcurso) REFERENCES CURSO (idcurso) );

```

Script DDL compatível com o estudo de caso Escola

ALUNO

O **Diagrama Entidade-Relacionamento (DER)** é uma ferramenta essencial para modelagem de banco de dados, oferecendo uma visão gráfica das relações entre entidades, atributos e suas interações.

A **entidade** representa um conjunto de objetos da realidade sobre os quais se deseja armazenar informações, sendo simbolizada por um retângulo com seu nome dentro. Por exemplo, em um sistema, uma entidade pode ser o conjunto de todos os alunos.

Exemplo de representação gráfica de entidade

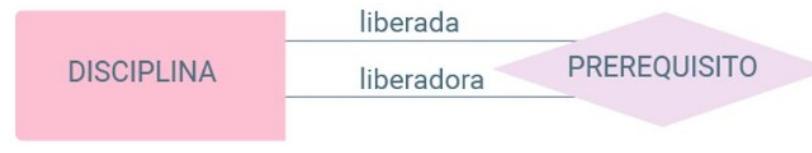


Exemplo de representação gráfica de relacionamento

O **relacionamento** descreve as associações entre as entidades e é representado por um losango, ligado por linhas às entidades envolvidas. Ele pressupõe a existência das entidades participantes, como no exemplo de um relacionamento chamado **POSSUI**.

Um **autorrelacionamento** ocorre quando uma entidade se relaciona consigo mesma. É importante distinguir os papéis de cada ocorrência dentro do relacionamento. Por exemplo, no caso de pré-requisitos para cursos, uma disciplina pode ser pré-requisito para outra.

No exemplo da **entidade DISCIPLINA**, o relacionamento **PREREQUISITO** conecta duas disciplinas: a **liberadora** (Cálculo I) e a **liberada** (Cálculo II). A disciplina liberadora é necessária antes da disciplina liberada.



Exemplo de representação gráfica de autorrelacionamento

A **cardinalidade** de relacionamentos expressa a quantidade mínima e máxima de ocorrências que uma entidade pode ter em relação a outra. No exemplo de **CURSO** e **DISCIPLINA**, as respostas às perguntas são:

- Uma disciplina não precisa estar associada a um curso (cardinalidade mínima 0).

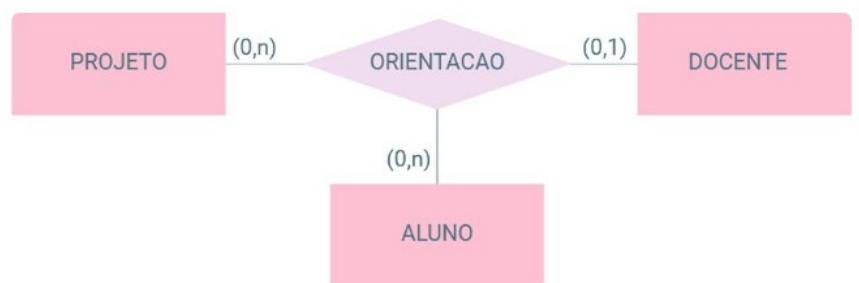
2. Uma disciplina pode estar associada a vários cursos (cardinalidade máxima N).
3. Um curso não precisa ter uma disciplina associada (cardinalidade mínima 0).
4. Um curso pode ter várias disciplinas (cardinalidade máxima N).

Este par (0,N) expressa a participação da entidade DISCIPLINA no relacionamento POSSUI



Relacionamento ternário envolve três entidades, como no caso de **projeto, aluno e docente**, formando uma associação entre os três.

Exemplo de representação gráfica do relacionamento POSSUI, com as cardinalidades definidas



O relacionamento **ORIENTACAO** envolve três entidades: **projeto, aluno e docente**. A cardinalidade para cada par de entidades é:

- **Docente** (cardinalidade máxima 1): Um aluno em um projeto pode ser orientado por no máximo um docente.
- **Aluno** (cardinalidade máxima N): Um docente pode orientar vários alunos em um projeto.
- **Projeto** (cardinalidade máxima N): Um aluno e um docente podem participar de vários projetos.

Exemplo de relacionamento ternário

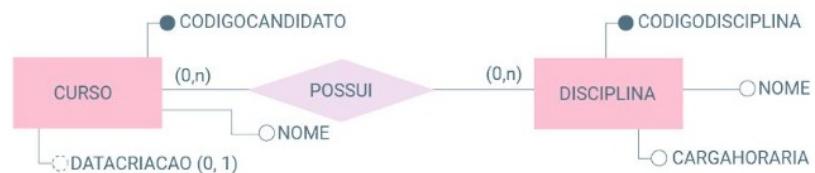
Atributo

Na modelagem de banco de dados, atributos representam características específicas de entidades e relacionamentos. Segundo Heuser (2009), um atributo é um dado associado a cada ocorrência de uma entidade ou relacionamento.

Exemplos de Atributos

- **Curso**: possui um código único, nome e, opcionalmente, data de criação.
- **Disciplina**: possui um código único, nome e carga horária.

No Diagrama Entidade-Relacionamento (DER), os atributos **CODIGOCURSO** e **CODIGODISCIPLINA** são identificadores únicos dentro de suas respectivas entidades.



Tipos Especiais de Atributos

- **Atributo Identificador**: usado para diferenciar cada ocorrência dentro de uma entidade. Sua representação gráfica inclui um traço com uma extremidade em círculo preenchido.
- **Atributo Opcional**: não obrigatório, indicado no DER por um traço com uma extremidade em círculo pontilhado.

Os demais atributos são obrigatórios.

Cardinalidade em Atributo

No Diagrama Entidade-Relacionamento (DER), a cardinalidade define as restrições de um atributo:

Cardinalidade Mínima	Cardinalidade Máxima	Significado
0	1	Opcional, Monovalorado
0	N	Opcional, Multivalorado
1	1	Obrigatório, Monovalorado
1	N	Obrigatório, Multivalorado

A maioria dos atributos em um DER são **monovalorados e obrigatórios**. Para facilitar a leitura, a cardinalidade (1,1) geralmente não é representada. Assim, se um atributo não exibir cardinalidade, considere-o **obrigatório e monovalorado**.

Atributo Composto

Atributos compostos podem ser subdivididos em partes menores. Por exemplo, o atributo **endereço** pode conter **logradouro, complemento, CEP e cidade**.

Modelo de Entidade e Relacionamento Estendido

Esse modelo adiciona novos elementos semânticos, incluindo **especialização, generalização e entidade associativa**.

Especialização/Generalização

A especialização e a generalização são mecanismos que aumentam a flexibilidade da modelagem de dados.

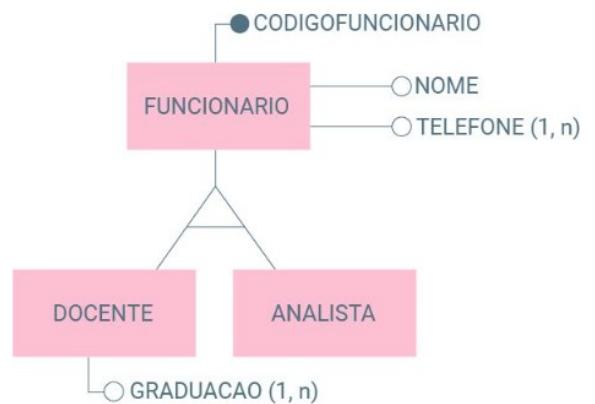
- **Generalização** identifica semelhanças entre entidades, agrupando-as em um nível mais abstrato.
- **Especialização** detalha características únicas de subconjuntos dessas entidades.

Por exemplo, uma instituição pode ter funcionários em geral, mas também categorias especializadas como **docentes** e **analistas**. Um **funcionário** possui **código único, nome e pelo menos um telefone**. A entidade **DOCENTE**, além de herdar essas propriedades, possui o atributo obrigatório **graduação**.

No **Diagrama Entidade-Relacionamento (DER)**, esse mecanismo é representado por um **triângulo**, onde a entidade mais genérica fica no topo e as especializadas abaixo.

A **herança de propriedades** garante que todas as entidades especializadas mantenham os atributos da entidade genérica.

Assim, todo docente herda as propriedades de funcionário, preservando a hierarquia de dados.



Classificações para Especialização/Generalização

A especialização/generalização pode ser classificada de duas formas:

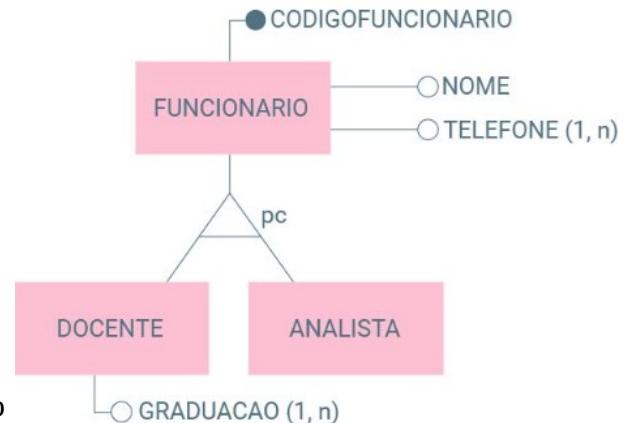
1. **Total ou Parcial** – Determina se todas as instâncias da entidade genérica devem pertencer a pelo menos uma entidade especializada:

- **Total (T)**: Todo funcionário deve ser docente ou analista.
- **Parcial (P)**: Pode existir funcionário sem especialização.

2. **Exclusiva ou Compartilhada** – Define se uma instância pode pertencer a múltiplas entidades especializadas:

- **Exclusiva (X)**: Um funcionário pode ser **apenas** docente ou analista.
- **Compartilhada (C)**: Um funcionário pode ser **ambos** docente e analista.

No exemplo apresentado, a especialização é **parcial e compartilhada (PC)**, pois pode haver funcionários não especializados e aqueles que atuam simultaneamente como **docente e analista**.



Entidade Associativa

A **entidade associativa** permite capturar informações adicionais sobre relacionamentos complexos entre entidades, enriquecendo a modelagem de dados.

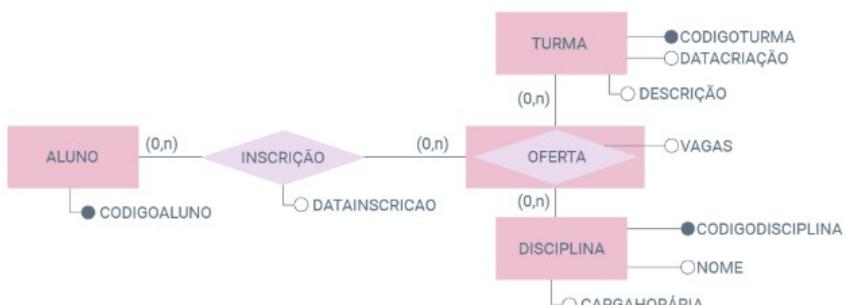
Por exemplo, ao modelar a inscrição de alunos em disciplinas, criamos a entidade **TURMA**, que possui **código, descrição e data de criação**. Cada turma pode estar associada a várias disciplinas, e uma disciplina pode ser ofertada em diversas turmas. Para organizar essa relação, usamos a entidade associativa **OFERTA**, que conecta **TURMA** e **DISCIPLINA**, além de incluir atributos como **número de vagas**.

A entidade **OFERTA** também se relaciona com **ALUNO** por meio do relacionamento **INSCRIÇÃO**, registrando quando o aluno se inscreveu na disciplina ofertada.

Perspectivas da Entidade Associativa

1. **Relacionamento** – **OFERTA** contém o atributo **VAGAS**, essencial para o planejamento das turmas.
2. **Entidade** – **OFERTA** permite identificar **qual turma e disciplina o aluno escolheu** ao se inscrever.

No Diagrama Entidade-Relacionamento (DER), a **entidade associativa** é representada por um **losango dentro de um retângulo**.



DER na Prática – Estudo de Caso: Empresa de Limpeza

A empresa **Serviços Domésticos** deseja um sistema para gerenciar a **alocação de empregados aos serviços solicitados pelos clientes**.

Fluxo do Processo

1. Cadastro de Clientes

- O cliente telefona para solicitar um serviço.
- A atendente verifica se ele já está cadastrado; se não, realiza o cadastro:
 - **Pessoa Jurídica:** CNPJ, razão social, endereço, telefone.
 - **Pessoa Física:** CPF, nome, endereço, telefone.
- A empresa atribui um **código único** a cada cliente.

2. Pedido de Serviço

- A atendente registra o pedido com:
 - Nome do cliente, data da abertura, data do serviço, local e descrição do serviço com metragem.
- Consulta uma **tabela de serviços** (código, descrição, valor por m², duração por m²).
- Calcula e informa ao cliente o **valor total e a duração do serviço**.

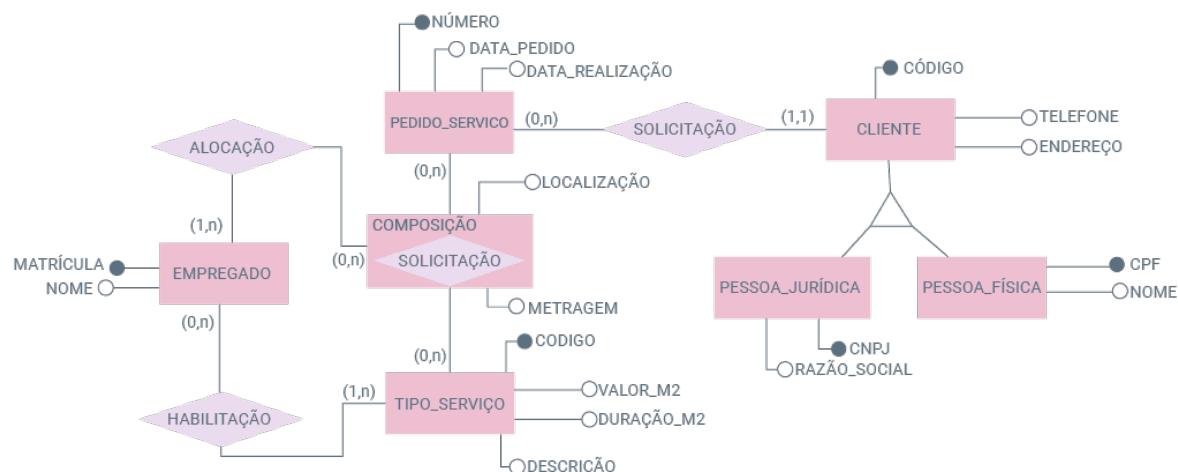
3. Alocação de Empregados

- Todo empregado pode executar **pelo menos um tipo de serviço**.
- Só pode ser alocado em serviços para os quais está habilitado.
- Cada empregado realiza **apenas um serviço por pedido**.

Roteiro para Modelagem Conceitual

1. Identificar entidades.
2. Definir entidades tipo e representá-las no DER.
3. Identificar e definir relacionamentos.
4. Estabelecer e representar cardinalidades.
5. Verificar extensões como especialização/generalização e entidades associativas.
6. Identificar e representar atributos.

Esse processo permite estruturar um **Diagrama Entidade-Relacionamento (DER)** eficiente, garantindo que todas as informações do sistema sejam bem organizadas e representadas.



Processo de Modelagem de Entidades e Relacionamentos

A modelagem de entidades e relacionamentos define as conexões essenciais entre entidades, estruturando o design do banco de dados.

Objetivos do DER

- Capturar as informações essenciais do negócio.
- Garantir um entendimento comum entre os profissionais.
- Facilitar a manutenção e aprendizado do sistema.

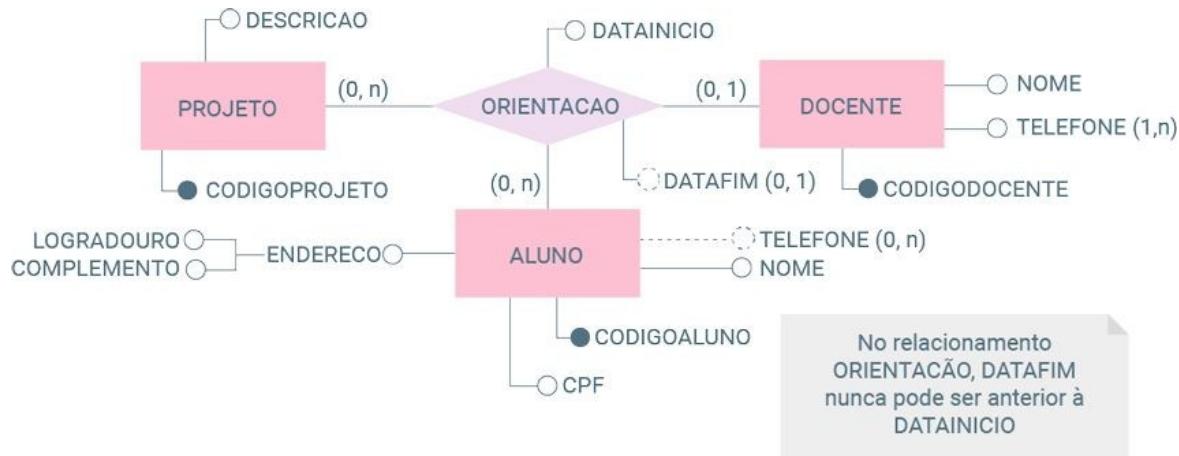
Restrições de Integridade

- Algumas restrições podem ser expressas no DER, como cardinalidade.
- Outras, conhecidas como **restrições semânticas**, precisam ser descritas separadamente em linguagem natural.

Exemplo: No relacionamento entre docentes e alunos em projetos, a **data de término da orientação nunca deve ser menor que a data de início**. Essa restrição é especificada em texto no DER.

Foco do DER

O principal objetivo do DER é estruturar um **banco de dados eficiente**, e não registrar todas as regras de integridade, que podem ser documentadas de outras formas.



Modelagem de Entidades e Relacionamentos

Ao modelar um banco de dados, diferentes representações podem resultar no mesmo esquema. Segundo Heuser (2009), dois modelos ER são equivalentes quando produzem a mesma estrutura final. Um exemplo disso é a transformação de um **relacionamento N:N** em uma entidade.

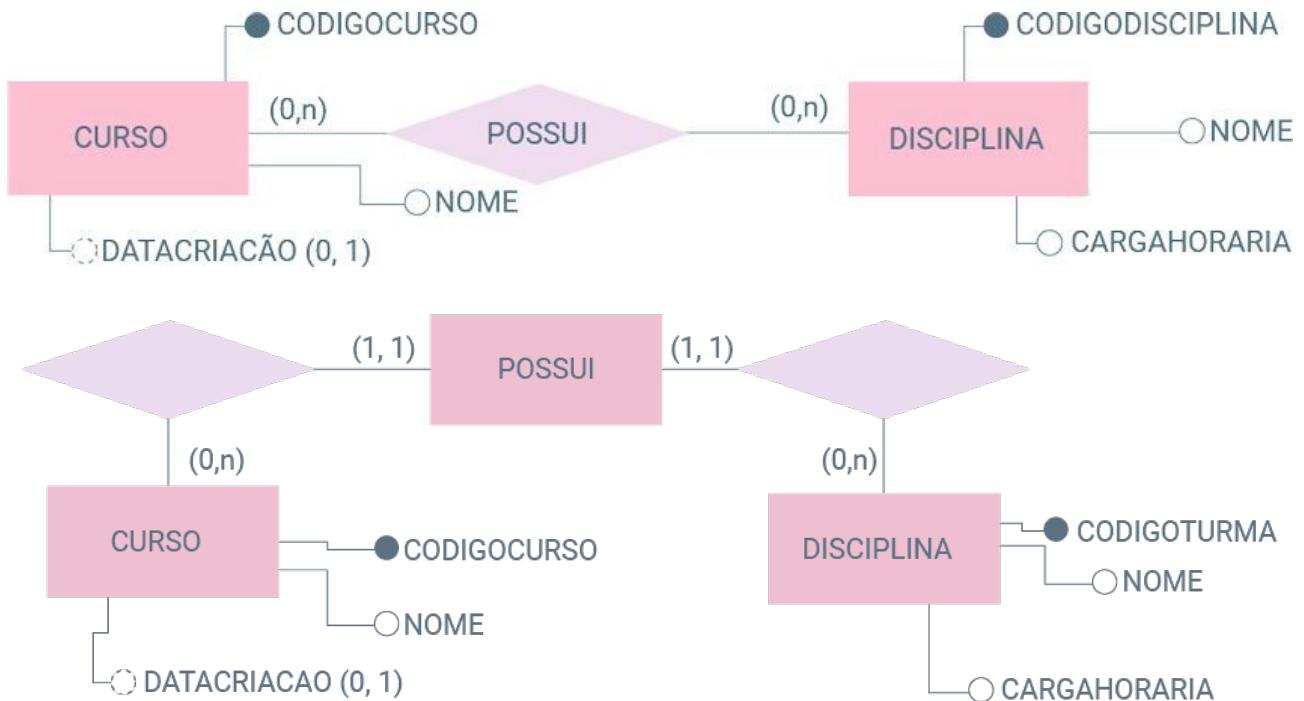
Transformação de Relacionamento N:N em Entidade

Para substituir um relacionamento N:N por uma entidade, seguimos estas etapas:

1. **Criar uma entidade** representando o relacionamento original. No exemplo, o relacionamento **POSSUI** entre **CURSO** e **DISCIPLINA** é transformado em uma entidade chamada **POSSUI**.

2. Relacionar a nova entidade com as entidades participantes do relacionamento original.
3. Adicionar atributos, caso existam, que pertenciam ao relacionamento original. Se não houver atributos, a entidade é criada sem eles.
4. Definir a chave identificadora da nova entidade com base nos relacionamentos com as entidades originais. No modelo, essa relação é representada por **linhas mais espessas**.
5. Estabelecer a cardinalidade (1,1) da nova entidade em relação às entidades originais, garantindo que cada ocorrência dela dependa da existência de ambas.

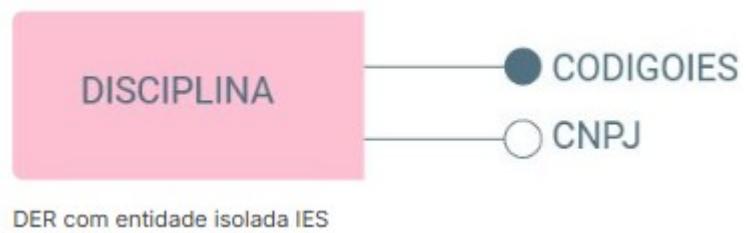
A conversão preserva a integridade do modelo, garantindo que um relacionamento só exista se ambas as entidades associadas também existirem. Essa dependência caracteriza a entidade criada como **entidade fraca**, pois sua existência depende das entidades às quais está vinculada.



Modelagem de Entidade Isolada

No DER modelado para uma única IES, não é necessário registrar a qual instituição alunos e cursos pertencem. Segundo Heuser (2009), uma **entidade isolada** não possui relacionamento com outras entidades.

No contexto do DER, a entidade **INSTITUICAO** pode ser usada para armazenar características da IES, como **código e CNPJ**. Caso seja necessário adicionar mais informações, como **data de criação, nome e telefone**, basta incluir novos atributos. Como o banco de dados é específico para uma única IES, essa entidade permanece **isolada**.



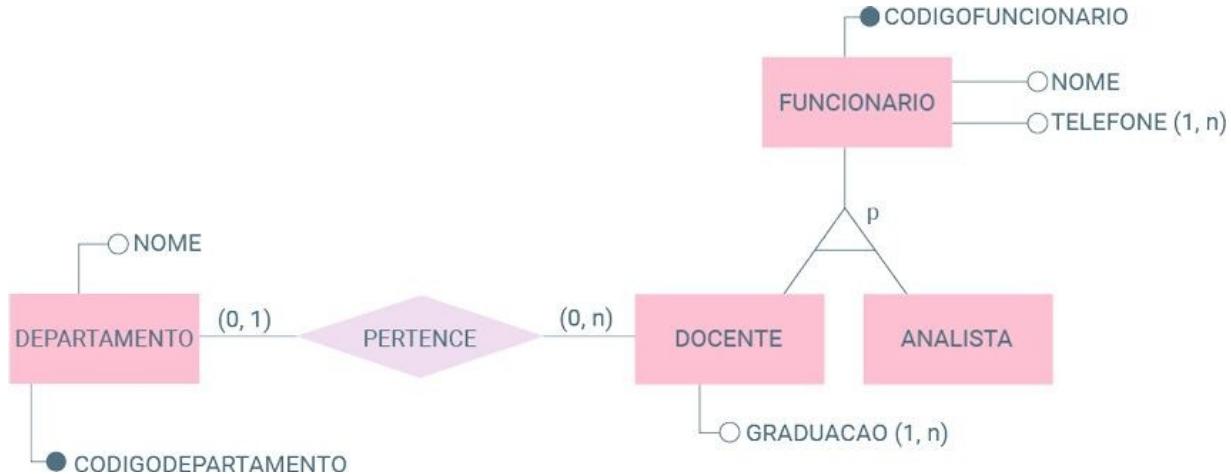
Quando Manter Histórico

A decisão de manter históricos no banco de dados é estratégica, impactando auditoria, análises temporais e conformidade.

No modelo acadêmico, surgiu a necessidade de associar **docentes a departamentos**, um requisito comum em IES. Para isso, foram definidos:

- **Departamento:** identificado por **código** e **nome**.
- **Docente:** pode estar associado a **no máximo um departamento**.

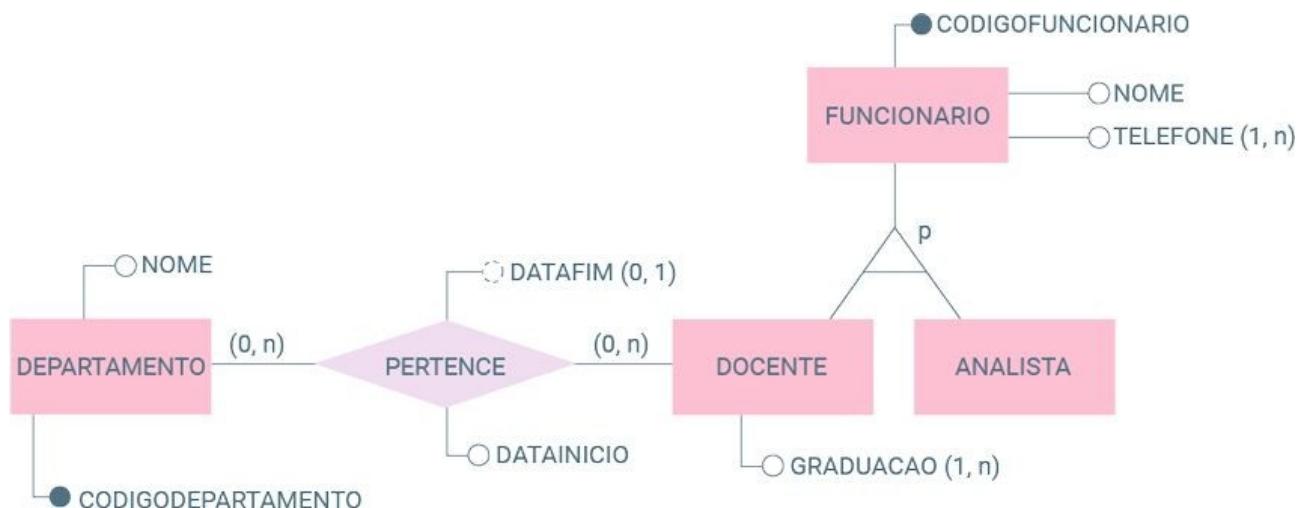
O DER construído atende a esses requisitos, com a entidade **DEPARTAMENTO** armazenando as informações e a **cardinalidade máxima 1** garantindo que cada docente pertença a um único departamento.



Histórico de Movimentação de Docentes

Requisitos de dados em projetos de banco de dados podem mudar, exigindo ajustes no modelo. Um novo requisito determina o armazenamento da **movimentação de docentes entre departamentos**, com **data de início e fim** da atuação.

O novo **DER** reflete essa mudança, mantendo a estrutura original, mas alterando a **cardinalidade máxima para N**, indicando que um docente pode atuar em vários departamentos ao longo da carreira. Além disso, foram adicionados dois atributos ao relacionamento para registrar as datas de movimentação.



Modelagem Avançada de Entidades e Relacionamentos

A criação de um **DER** não ocorre em uma única etapa, mas de forma **incremental**, sendo aprimorada conforme novos requisitos surgem. Esse refinamento contínuo permite representar com precisão as interações no banco de dados.

A abordagem utilizada segue a **estratégia descendente**, onde conceitos mais abstratos são modelados inicialmente e, posteriormente, detalhados. Geralmente, começamos identificando entidades e, em seguida, definimos atributos e relacionamentos.

Etapas do Processo:

- Modelo Inicial:** Identificação de entidades, relacionamentos, especializações e cardinalidade máxima, além do mapeamento de atributos e identificadores.
- Modelo Detalhado:** Definição das cardinalidades dos relacionamentos e identificação de outras restrições de integridade.
- Validação:** Revisão do modelo junto ao usuário para garantir que ele atenda aos requisitos do sistema.

Esse processo não é rígido, permitindo revisões e retornos a etapas anteriores conforme necessário, garantindo que o modelo reflita fielmente a estrutura e as regras do banco de dados.

Modelagem de Entidade e Relacionamento na Prática

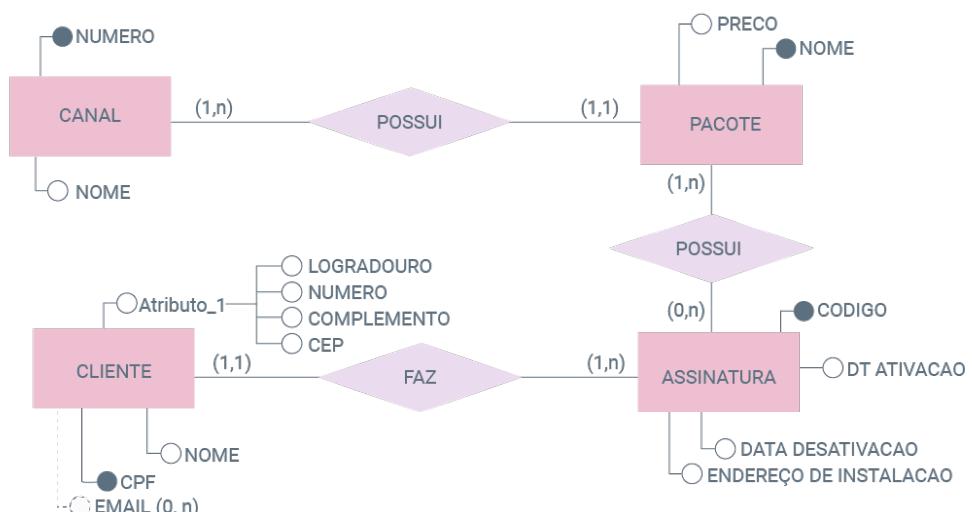
Neste estudo de caso, a empresa de TV a cabo deseja automatizar seu serviço de cobrança, com base nos seguintes requisitos:

- Canais:** Identificados por número e nome.
- Pacotes:** Conjunto de canais com nome exclusivo, preço e ao menos um canal. Cada canal pertence a apenas um pacote.
- Clientes:** Devem ter CPF, nome, endereço e e-mail. Um cliente pode ter vários e-mails.
- Assinaturas:** Associadas a um cliente e um ou mais pacotes, contendo data de ativação, desativação e endereço de instalação.

Roteiro de Prática:

- Identificar entidades** e representá-las no **DER**.
- Identificar** relacionamentos e definir suas cardinalidades.
- Modelar atributos** das entidades.

O modelo descritivo deve ser transformado em **DER** seguindo essas etapas para representar adequadamente o sistema.

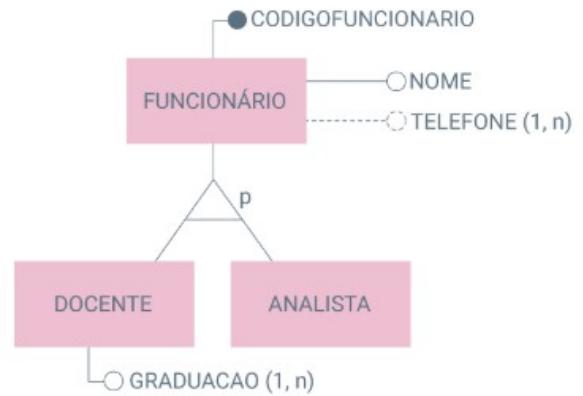


Atributo X Entidade

A decisão entre usar uma **entidade** ou um **atributo** é fundamental para a construção de bancos de dados, impactando a estrutura, eficiência e flexibilidade do modelo.

Embora os exemplos anteriores tenham usado atributos para caracterizar entidades e relacionamentos, nem sempre isso é a melhor abordagem. Em alguns casos, modelar um objeto como atributo pode ser inadequado.

No exemplo acadêmico, todo docente precisa ter pelo menos uma **graduação** registrada. Isso pode ser modelado de diferentes formas, dependendo das necessidades do sistema.



Atributo vs Entidade e Cardinalidades

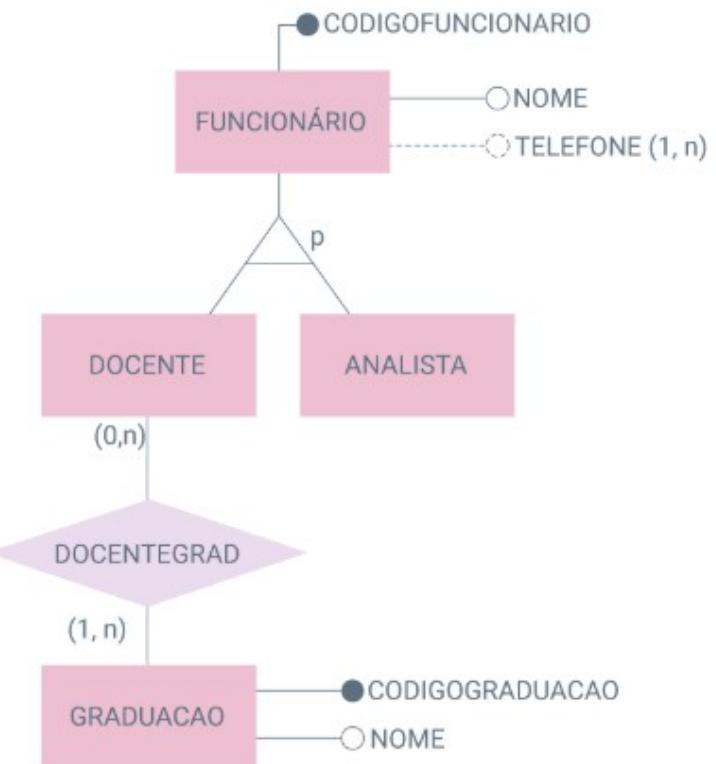
Ao decidir entre modelar um objeto como **atributo** ou **entidade**, é fundamental entender as **cardinalidades** e as implicações no modelo.

- **Cardinalidade mínima 1:** Indica que a informação é obrigatória para o registro de um dado.
- **Cardinalidade máxima N:** Permite registrar múltiplos valores associados ao mesmo dado, como vários cursos para um docente.

Por exemplo, quando a graduação de um docente é mapeada como **atributo**, não faz sentido inserir várias vezes o nome do curso no sistema, já que isso poderia gerar representações duplicadas (como "Ciência da Computação"). Se fosse necessário incluir o **ano de formatura** do docente, o mais adequado seria modelar a graduação como **entidade**. Isso é refletido no novo modelo, onde a graduação pode ser registrada sem um vínculo direto com o docente (cardinalidade mínima 0), mas cada docente deve ter ao menos uma graduação (cardinalidade mínima 1).

Além disso, caso fosse necessário armazenar informações como o **ano de término** da graduação, isso poderia ser representado como um **atributo** no relacionamento entre as entidades **DOCENTE** e **GRADUAÇÃO**.

Finalmente, ao decidir entre **atributo** ou **especialização**, a regra é simples: quando um objeto possui **atributos ou relacionamentos adicionais**, a escolha é pela **especialização**. No exemplo dado, a **entidade FUNCIONARIO** foi especializada, alinhando-se a esse critério, e a entidade **DOCENTE** foi relacionada à **GRADUAÇÃO**, validando a escolha de especialização.

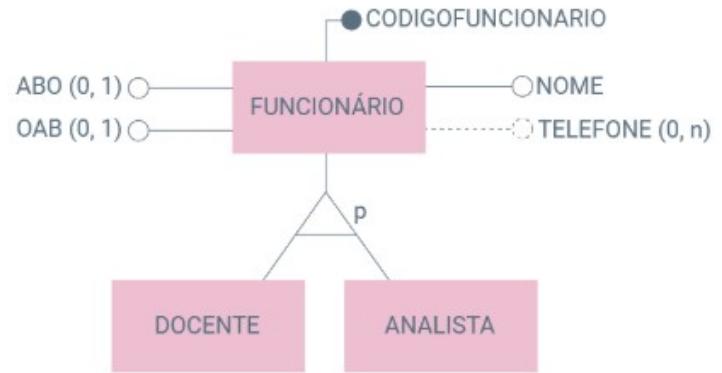


Atributo Opcional

Na modelagem de bancos de dados, a decisão de tornar um atributo **obrigatório** ou **opcional** é crucial, pois define a rigidez ou flexibilidade do sistema, impactando a **integridade dos dados**.

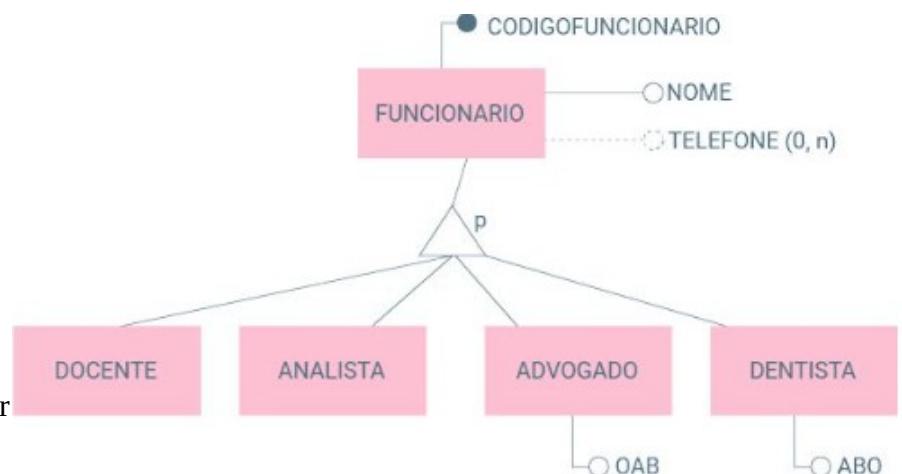
Em certos casos, vários atributos podem ser **opcionais** dentro de uma entidade. Isso exige uma análise para verificar se esses atributos sugerem a necessidade de **entidades especializadas**.

Por exemplo, ao precisar saber se um funcionário está registrado na **OAB** ou na **ABO**, a modelagem pode ser adaptada para refletir essa nova exigência, possivelmente criando novas entidades ou ajustando a estrutura existente.



No DER, os atributos **OAB** e **ABO** foram adicionados à entidade **FUNCIONARIO** como opcionais. No entanto, não está claro quais combinações de atributos são válidas (por exemplo, um funcionário pode ter ambos?).

Após análise, percebe-se que **OAB** e **ABO** representam categorias distintas (Advogados e Odontólogos). Por isso, foi decidido modelá-los como **entidades especializadas** de **FUNCIONARIO**, o que reflete melhor a realidade e evita atributos opcionais na entidade **FUNCIONARIO**.



Na modelagem de bancos de dados, a decisão entre **atributo multivlorado** e **monovalorado** é crucial. Um atributo **multivlorado** permite múltiplos valores para uma entidade.

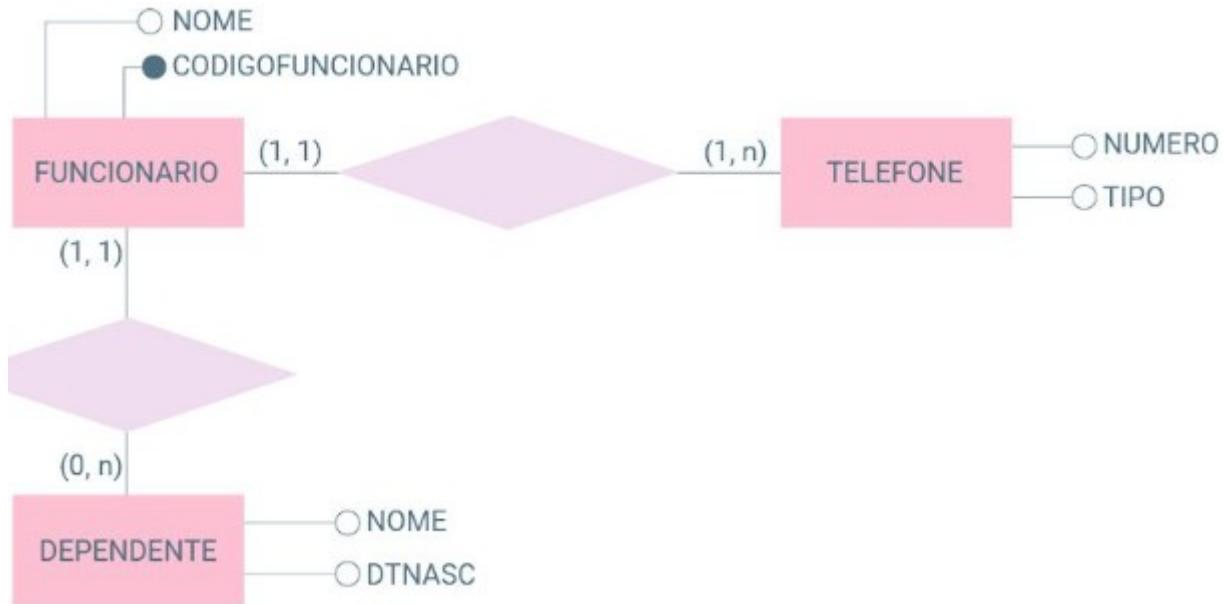
Suponha que seja necessário modelar que um **funcionário** pode ter **dependentes**. Uma primeira abordagem seria adicionar um **atributo opcional** na entidade **FUNCIONARIO**.



A modelagem com atributos **multivlorados** deve ser evitada por dois motivos principais:

1. Não há uma implementação direta para atributos multivlorados em um **SGBD relacional**.
2. Frequentemente, esses atributos ocultam informações importantes, como **atributos** e **relacionamentos**.

Para melhorar a modelagem, deve-se usar entidades separadas, como para **dependentes** e **telefones** de um **funcionário**. Isso torna o modelo mais preciso, legível e flexível, permitindo a adição de novos atributos de forma mais clara.

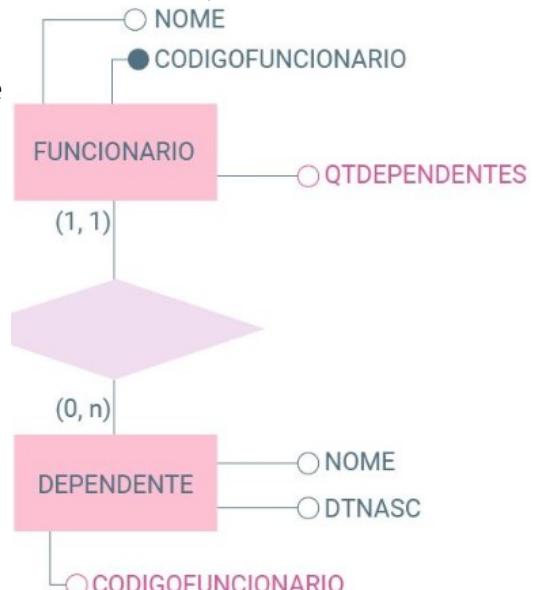


Atributos redundantes e compostos

Neste tópico, discutimos os trade-offs entre simplificação da estrutura e eficiência, visando uma modelagem de dados coesa e eficaz.

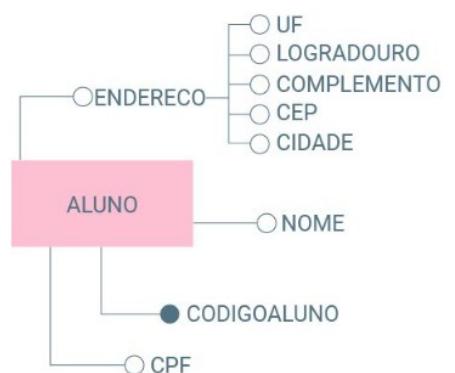
Atributos redundantes são aqueles derivados de procedimentos de busca ou cálculos no banco de dados. Um exemplo seria a quantidade de dependentes de um funcionário, que pode ser calculada a partir de um relacionamento, e o número de matrícula do funcionário, que pode ser obtido pelo relacionamento com a entidade FUNCIONARIO.

Em um Diagrama de Entidade-Relacionamento (DER), atributos redundantes, como QTDEDEPENDENTES na entidade FUNCIONARIO ou CODIGOFUNCIONARIO na entidade DEPENDENTE, devem ser evitados, pois suas informações podem ser derivadas dos relacionamentos. Atributos redundantes não devem aparecer no DER, pois o modelo não distingue entre atributos redundantes e não redundantes.

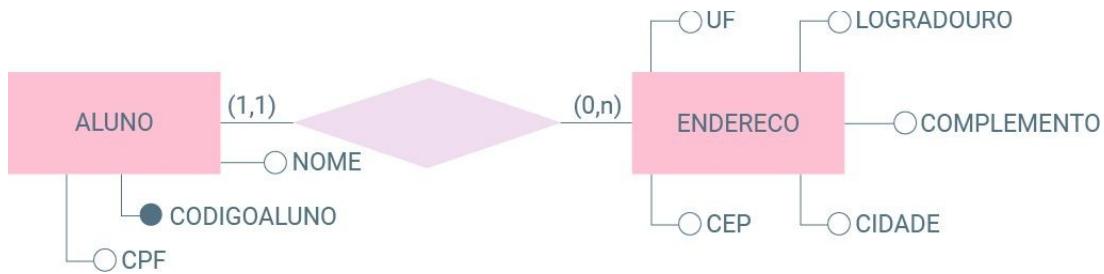


Atributo composto

Um atributo composto pode ser dividido em subpartes ou atributos básicos, cada um com significados próprios. No exemplo do DER, o endereço do aluno foi inicialmente modelado como um atributo composto, subdividido em logradouro e complemento. Ao adicionar mais atributos, o modelo se tornou mais denso visualmente. Uma



alternativa é modelar o atributo composto como uma entidade separada relacionada à entidade principal, como mostrado em um DER posterior.

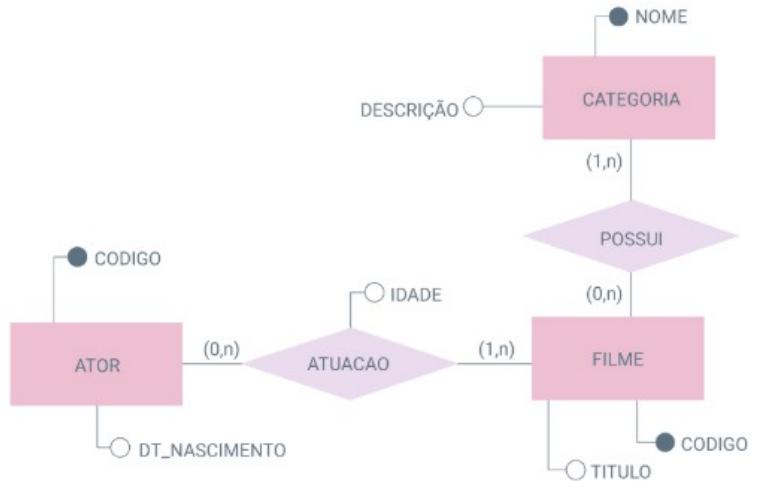


Modelagem de atributos na prática

Neste estudo de caso, o minimundo envolve filmes, com atributos como código, título e categoria (comédia, suspense, etc.). É necessário saber os atores principais de cada filme, sendo que nem todos os filmes os possuem, além de informações sobre o nome real e a idade de cada ator.

Para realizar a modelagem conceitual, siga os seguintes passos:

1. Identificar as entidades.
2. Definir e representar as entidades no DER.
3. Identificar os relacionamentos entre as entidades.
4. Definir e representar os relacionamentos no DER.
5. Estabelecer as cardinalidades dos relacionamentos.
6. Identificar os atributos e suas características.
7. Modelar os atributos, definindo se são:
 - Únicos ou não únicos.
 - Opcionais ou obrigatórios.
 - Simples ou compostos.
 - Monovalorados ou multivvalorados.



Modelo relacional e componentes básicos de uma tabela

O modelo relacional organiza dados em tabelas, atributos e relacionamentos, essenciais para a criação de sistemas de banco de dados eficientes.

No contexto acadêmico, "relação" é o termo usado para "tabela". O banco de dados é visto como uma coleção de relações.

Uma tabela é composta por tuplas (linhas) e atributos (colunas). As tuplas representam as instâncias de dados, enquanto as colunas, ou atributos, armazenam características específicas. Cada coluna tem um nome para facilitar a interpretação dos dados armazenados.



Nome de tabelas: Em bancos de dados relacionais, cada tabela deve ter um nome único e refletir claramente o objeto modelado. Por exemplo, a tabela **ALUNO** deve conter dados sobre alunos.

Colunas de tabelas: Cada coluna também deve ter nome único e representar uma única informação. As colunas são monovaloradas e atômicas (não podem ser compostas). Ao criar uma tabela, é necessário definir o tipo de dado (caractere, numérico, data, booleano) e se a coluna é opcional ou obrigatória. Valores opcionais podem ser nulos.

Linhos de tabelas: Cada linha representa um item de informação (exemplo: um aluno). Tabelas devem armazenar dados específicos para um único tipo de objeto, como **ALUNOS**, e não para outros, como disciplinas ou docentes.

Chave Primária: No banco de dados, a chave primária é usada para identificar exclusivamente cada linha de uma tabela. Ela pode ser **simples**, com uma única coluna, ou **composta**, envolvendo mais de uma coluna. A chave primária tem três propriedades principais: **unicidade** (não permite valores duplicados), **monovalorada** (permite apenas um valor por vez) e **obrigatória** (não pode ser nula).

- **Chave Primária Simples:** Exemplo: Na tabela de **ALUNO**, a coluna **CODIGOALUNO** pode ser escolhida como chave primária, pois cada código de aluno é único, garantindo a identificação do registro.
- **Chave Primária Composta:** Exemplo: Na tabela **DEPENDENTE**, a chave primária é composta pelas colunas **CODIGOFUNCIONARIO** e **NRDEPENDENTE**, pois nenhum dos dois campos isoladamente é suficiente para identificar unicamente um dependente (um funcionário pode ter vários dependentes, e um dependente pode ter o mesmo número em várias linhas).

Chave Mínima: A chave primária deve ser composta pelas colunas necessárias e suficientes para garantir a identificação única da linha. Não se refere à quantidade de colunas, mas ao fato de que cada coluna deve ser essencial para diferenciar os registros.

- **Chave Mínima Simples:** No caso da tabela **ALUNO**, a chave primária **CODIGOALUNO** é mínima, pois sozinha já é suficiente para distinguir os alunos. Se adicionássemos **CPF** à chave primária, ela deixaria de ser mínima, pois **CODIGOALUNO** já é suficiente para distinguir os registros.

Chave Candidata: Se houver mais de uma coluna capaz de identificar unicamente uma linha, como **CODIGOALUNO** e **CPF** na tabela **ALUNO**, essas colunas são chamadas de chaves candidatas. Qualquer uma delas poderia ser escolhida como chave primária.

Chave Alternativa: Depois de escolher a chave primária, as outras chaves candidatas tornam-se chaves alternativas. No exemplo da tabela **ALUNO**, **CPF** é uma chave alternativa, pois poderia ser usada em vez de **CODIGOALUNO**.

Autoincremento: Alguns SGBDs oferecem a opção de **autoincremento**, onde o valor da chave primária é gerado automaticamente quando um novo registro é inserido, garantindo a unicidade sem a necessidade de inserção manual.

Esse resumo mantém todos os conceitos importantes, abordando a definição e as características das chaves primária, simples, composta, mínima, candidata e alternativa.

Chave Estrangeira: A chave estrangeira é usada para conectar tabelas em um banco de dados relacional, garantindo a integridade referencial. Ela é uma coluna ou combinação de colunas cujo valor precisa corresponder a um valor de chave primária em outra tabela.

Relacionamento entre Tabelas: Relacionamentos entre tabelas são essenciais para o modelo relacional. Por exemplo, a tabela **DEPENDENTE** tem a coluna **CODIGOFUNCIONARIO** como chave estrangeira, referenciando a chave primária **CODIGOFUNCIONARIO** da tabela **FUNCIONARIO**. Isso assegura que cada dependente esteja associado a um funcionário, permitindo várias dependências por funcionário ou nenhum, dependendo do caso.

FUNCIONARIO				
CODIGOFUNCIONARIO	NOME	SEXO	CPF	DTNASCIMENTO
1	José Maciel	M	23456239999	13/09/1982
2	Pedro Antônio	M	98789345552	13/11/1986
3	Debora Silva	F	12332149048	15/02/1998
4	Amanda de Miranda	F	60989893223	15/05/1997

DEPENDENTE			
CODIGOFUNCIONARIO	NRDEPENDENTE	NOME	DTNASCIMENTO
1	1	Andrey Campos	13/07/2019
1	2	Manoel Oliveira	13/12/2018
2	1	João Silva	15/02/2017
2	2	José Maciel	15/07/2016

Restrição de Integridade Referencial: A chave estrangeira impõe regras para garantir a consistência dos dados:

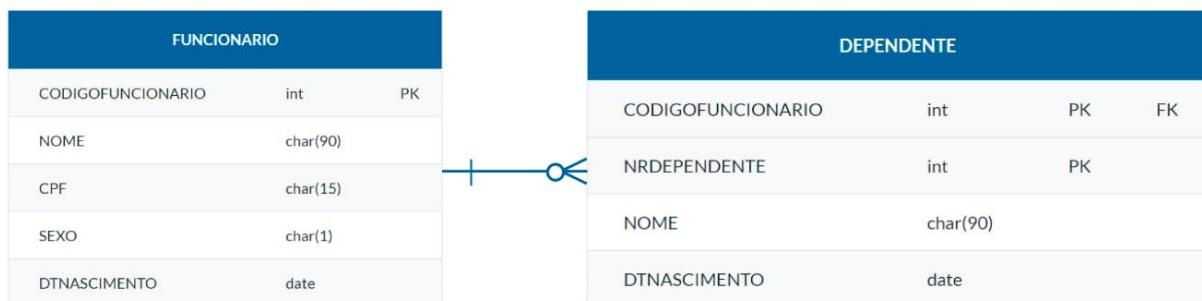
- Inclusão de Linha:** Ao inserir uma linha na tabela com chave estrangeira, o valor dessa chave deve existir na chave primária da tabela referenciada. Por exemplo, ao incluir um dependente, o valor de **CODIGOFUNCIONARIO** deve ser um dos valores existentes em **FUNCIONARIO**.

2. **Alteração da Chave Estrangeira:** Se o valor de uma chave estrangeira for alterado, o novo valor deve existir como chave primária na tabela referenciada. No exemplo, se o responsável de um dependente for alterado, o novo valor de **CODIGOFUNCIONARIO** precisa corresponder a um valor válido de **FUNCIONARIO**.
3. **Exclusão de Linha:** Não é possível excluir um registro de uma tabela que tenha referências em outra. Por exemplo, se um funcionário tem dependentes registrados, ele não pode ser excluído, pois isso quebraria a relação com a tabela **DEPENDENTE**.
4. **Alteração de Valor da Chave Primária:** Se o valor de uma chave primária for alterado, como no caso de **CODIGOFUNCIONARIO**, o novo valor precisa ser replicado nas linhas da tabela que referenciam essa chave, mantendo a consistência dos dados.

Essas restrições garantem a **integridade referencial**, ou seja, os valores das chaves estrangeiras precisam sempre corresponder a valores válidos na tabela referenciada, preservando a coesão e consistência dos dados no banco de dados.

Esquema Diagramático de Banco de Dados Relacional

O esquema diagramático é uma representação visual que mapeia a complexidade organizacional dos dados, revelando as interconexões entre tabelas, atributos e chaves. Essa representação não é apenas gráfica, mas funciona como um “mapa” da estrutura do banco de dados.



Notação “Pé de Galinha” (Crow's Foot):

- Cada tabela é ilustrada por um retângulo dividido em duas partes:
 - **Primeira divisão:** Exibe o nome da tabela.
 - **Segunda divisão:** Lista as colunas com informações sobre nome, tipo de dado e símbolos que indicam chave primária (PK) e chave estrangeira (FK).
- O símbolo em forma de “pé de galinha” junto à tabela **DEPENDENTE** indica o lado N do relacionamento entre as tabelas.

Esquema Textual:

- O banco de dados também pode ser descrito textualmente. Por exemplo:
 - **FUNCIONARIO** (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO)
 - **DEPENDENTE** (CODIGOFUNCIONARIO, NRDEPENDENTE, NOME, DTNASCIMENTO)

- A declaração “CODIGOFUNCIONARIO REFERENCIA FUNCIONARIO” indica que a coluna CODIGOFUNCIONARIO na tabela DEPENDENTE é uma chave estrangeira que referencia a chave primária da tabela FUNCIONARIO.
- No esquema textual, os nomes das tabelas são seguidos pela lista de colunas, onde as chaves primárias são sublinhadas e as chaves estrangeiras são declaradas usando o padrão “nomecoluna(s) referencia nometabela(s)”.

Normalização de Banco de Dados

A normalização visa organizar os dados de forma eficiente, reduzindo redundâncias e anomalias para garantir consistência e integridade. Esse processo é essencial para criar bancos de dados flexíveis e otimizados.

Objetivo da Normalização:

Ela avalia se um banco de dados foi bem projetado e pode ser aplicada em qualquer representação de dados, como telas de sistemas ou relatórios.

Processo de Normalização:

O processo segue as formas normais (FN) que são regras para garantir que as tabelas estejam bem estruturadas. As formas normais mais comuns são 1FN, 2FN, 3FN, e a normalização geralmente é realizada até a 3FN.

Etapas da Normalização até a 3FN:

1. Identificar a origem dos dados.
2. Criar a tabela não normalizada a partir dos dados.
3. Aplicar as regras da 1FN.
4. Aplicar as regras da 2FN.
5. Aplicar as regras da 3FN.

Exemplo de Relatório de Alocação Docente:

O relatório lista docentes alocados a projetos de pesquisa com informações como código do projeto, tipo, descrição, nome, categoria, salário, data de início e tempo alocado.

Tabela Não Normalizada:

A tabela não normalizada (PROJETO) inclui dados de projetos e docentes alocados a eles, mas contém redundâncias. A coluna **DOCENTE** é composta por várias colunas e repete informações como nome e salário para o mesmo docente em projetos diferentes, o que caracteriza a redundância.

Objetivo do Processo:

Transformar a tabela não normalizada em uma estrutura que elimine redundâncias e siga as formas normais, começando pela 1FN.

Primeira Forma Normal (1FN)

A 1FN visa eliminar atributos multivalorados ou compostos, garantindo que cada atributo tenha valores atômicos. Isso promove a uniformidade e a estruturação dos dados, facilitando consultas e manipulações.

Condição de 1FN:

Uma tabela está na 1FN quando não contém atributos multivalorados ou compostos.

Passos para aplicar a 1FN:

1. Criar uma tabela com a chave primária da tabela não normalizada.
 2. Assegurar que todas as colunas sejam atômicas.

Exemplo de Tabelas em 1FN:

- **PROJETO:** Contém informações sobre projetos, com colunas como código, tipo e descrição.
 - **PROJETODOCENTE:** Relaciona docentes aos projetos, com informações detalhadas sobre cada docente.

Dependência Funcional:

Na tabela **PROJETODOCENTE**, o nome do docente depende do código do docente, formando uma dependência funcional. Ou seja, **CODIGODOCENTE → NOME**.

Dependência Funcional Parcial:

A relação **CODIGOPROJETO, CODIGODOCENTE** → **NOME** indica que o nome do docente depende apenas do código do docente, não necessitando do par completo para determinar o nome. Isso caracteriza uma dependência funcional parcial, onde um atributo depende apenas de parte da chave primária composta.

Segunda Forma Normal (2FN)

A 2FN visa eliminar dependências funcionais parciais, aprimorando a estrutura dos dados. Para uma tabela estar na 2FN, ela deve estar na 1FN e não ter dependências parciais.

Condição de 2FN:

A tabela deve estar na 1FN e não apresentar dependências parciais, onde um atributo depende apenas de parte da chave primária composta.

Passos para aplicar a 2FN:

1. Manter tabelas com chave primária simples.
 2. Identificar dependências parciais.
 3. Criar tabelas para cada dependência parcial.

Exemplo de Tabelas na 2FN:

- **PROJETO:** Contém informações sobre o projeto (código, tipo, descrição).
 - **PROJETODOCENTE:** Relaciona projetos aos docentes, com código do projeto, código do docente, data de início e tempo alocado.
 - **DOCENTE:** Contém dados dos docentes (código, nome, categoria, salário).

A tabela **PROJETODOCENTE** foi ajustada para remover dependências parciais, separando as informações dos docentes em uma tabela própria (DOCENTE). Assim, o modelo está na 2FN, pois não há mais dependências parciais.

PROIETO DOCENTE

Terceira Forma Normal (3FN)

A 3FN visa eliminar dependências transitivas, garantindo que todo atributo esteja diretamente relacionado à chave primária, evitando redundâncias e melhorando a consulta e manutenção dos dados.

Dependência Funcional Transitiva:

Acontece quando uma coluna não chave depende de outra não chave. Exemplo: Na tabela DOCENTE, o salário depende da categoria, e a categoria não faz parte da chave primária.

Condição de 3FN:

A tabela deve estar na 2FN e não ter dependências transitivas.

Passos para aplicar a 3FN:

1. Manter tabelas com menos de duas colunas não chave.
2. Identificar dependências transitivas.
3. Criar tabelas para essas dependências.

Exemplo de Tabelas na 3FN:

- **PROJETO**: Contém código, tipo e descrição do projeto.
- **PROJETODOCENTE**: Relaciona projetos a docentes com códigos, data de início e tempo alocado.
- **DOCENTE**: Contém dados dos docentes (código, nome, categoria).
- **CATEGORIA**: Relaciona categoria e salário dos docentes.

Após a aplicação da 3FN, o modelo está livre de dependências transitivas e segue um formato mais eficiente para consultas e manutenção.

PROJETODOCENTE

CODIGODOCENTE	NOME	CATEGORIA	SALARIO

Normalização na prática

A normalização de um sistema de controle de vendas envolve transformar as tabelas para as formas normais 1FN, 2FN e 3FN, a partir de um arquivo de notas fiscais com dados como: número da NF, cliente, mercadorias e total da venda.

Processos para obtenção das formas normais:

- **Primeira Forma Normal (1FN):**
 - Eliminar grupos repetitivos e atributos compostos.
 - Definir chaves candidatas e escolher a chave primária.
 - Tornar atributos compostos atômicos e itens multivvalorados em outras tabelas.
- **Segunda Forma Normal (2FN):**
 - Remover dependências parciais em tabelas com chaves compostas.
 - Identificar colunas não chave e verificar se dependem total ou parcialmente da chave.

- Criar tabelas para as dependências parciais e excluir colunas dependentes da chave.
 - **Terceira Forma Normal (3FN):**
 - Eliminar dependências transitivas (colunas não chave dependentes de outras colunas não chave).
 - Identificar colunas que dependem de outras não chave e criar tabelas separadas para elas.

Esses processos garantem que as tabelas estejam em conformidade com as normas de normalização, eliminando redundâncias e melhorando a estrutura do banco de dados.

Mapeamento conceitual-lógico de entidades

O mapeamento conceitual-lógico de entidades conecta o modelo conceitual à implementação no modelo lógico. Ele transforma entidades e relacionamentos do modelo conceitual em estruturas de dados relacionais para facilitar a manipulação e armazenamento de informações.

No desenvolvimento de um banco de dados, o Diagrama de Entidade e Relacionamento (DER) é utilizado para criar o modelo conceitual, onde entidades e seus atributos são definidos. A partir disso, são aplicadas regras formais para transformar o modelo conceitual em um modelo lógico, servindo de base para a implementação de um banco de dados relacional.

Regras de mapeamento

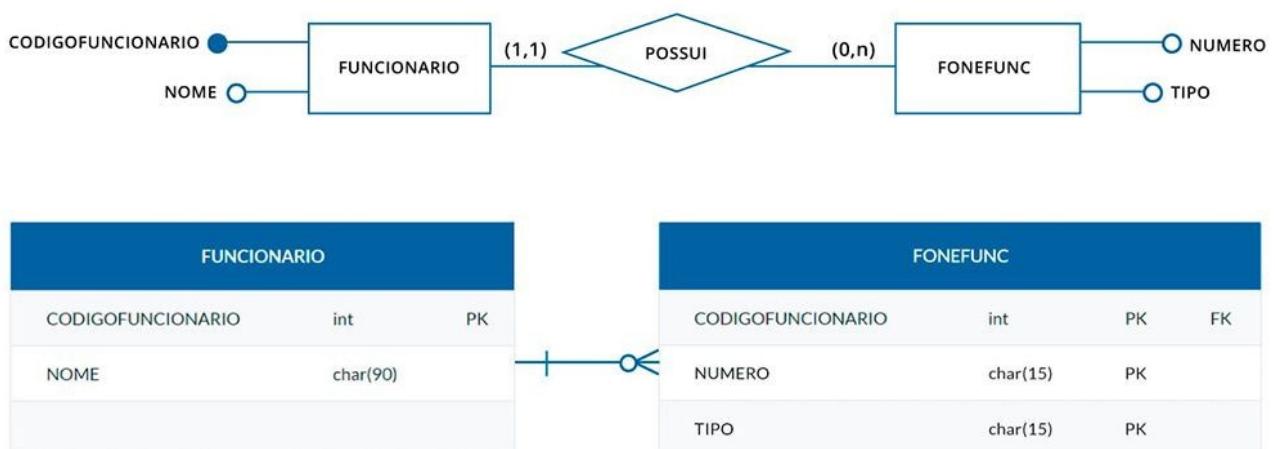
O mapeamento conceitual-lógico é dividido em quatro etapas:

1. Entidades
 2. Relacionamentos
 3. Atributos multivalorados
 4. Especialização/generalização

Mapeamento de entidades:

- **Entidade forte ou independente**
 - **Entidade fraca ou dependente**

Exemplo: No DER, temos as entidades **FUNCIONARIO** (forte) e **FONEFUNC** (fraca).



A tabela **FUNCIONARIO** aplica a regra para **entidade forte** e **FONEFUNC** aplica a regra para **entidade fraca**.

Após o mapeamento, a representação gerada é:

- **FUNCIONARIO** (CODIGO FUNCIONARIO, NOME)
- **FONEFUNC** (CODIGO FUNCIONARIO, NUMERO, TIPO)
CODIGO FUNCIONARIO **referencia** FUNCIONARIO.

O mapeamento de relacionamentos transforma as interconexões entre entidades do modelo conceitual em estruturas de dados relacionais, considerando diferentes tipos de cardinalidade.

Para relacionamentos 1:1:

- Cardinalidade (0,1):(0,1): Adicionar coluna(s) ou criar tabela própria.
- Cardinalidade (0,1):(1,1): Fusão de tabelas ou adição de colunas.
- Cardinalidade (1,1):(1,1): Fusão de tabelas.

Para relacionamentos 1:n:

- Identificar a tabela do lado N.
- Adicionar chave estrangeira à tabela do lado N.
- Cada atributo simples vira uma coluna.

Para relacionamentos n:n:

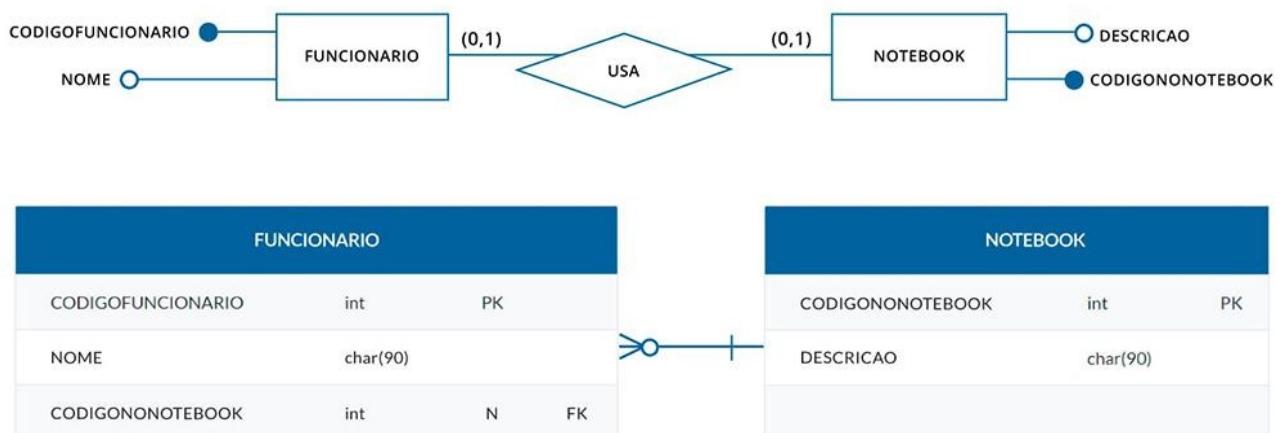
- Criar tabela T.
- Tabela T tem chaves estrangeiras das tabelas participantes.
- Tabela T tem chave primária composta.
- Cada atributo simples vira uma coluna.

Para relacionamentos n-ários (semelhante a n:n):

- Criar tabela T.
- Tabela T tem chaves estrangeiras e chave primária composta.
- Cada atributo simples vira uma coluna.

Exemplo de mapeamento 1:1:

No relacionamento 1:1 entre **FUNCIONARIO** e **NOTEBOOK**, com cardinalidade (0,1):(0,1), pode-se adicionar colunas ou criar uma tabela separada.

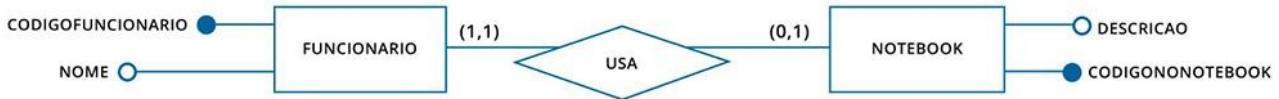


No relacionamento 1:1 com cardinalidade (0,1):(1,1), priorizou-se a adição de uma coluna, criando a coluna **CODIGONOTEBOOK** como chave estrangeira na tabela **FUNCIONARIO**. A coluna é opcional.

Modelo resultante:

- **NOTEBOOK** (CODIGONOTEBOOK, DESCRICAO)
- **FUNCIONARIO** (CODIGOFUNCIONARIO, NOME, CODIGONOTEBOOK)
- **CODIGONOTEBOOK REFERENCIA NOTEBOOK**

O DER mostra o relacionamento 1:1 com participação obrigatória e opcional.



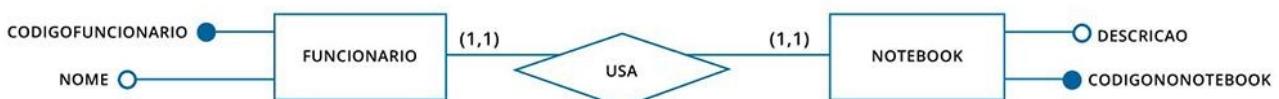
Modelo lógico gerado:

- **FUNCIONARIO**
 - CODIGOFUNCIONARIO (int, PK)
 - NOME (char(90))
 - CODIGONOTEBOOK (int, N)
 - DESCRICAO (char(90), N)

Esse modelo é resultado de um relacionamento 1:1, onde uma entidade tem participação obrigatória e a outra, opcional. Para esse tipo de relacionamento, optou-se pela fusão de tabelas, com as colunas **CODIGONOTEBOOK** e **DESCRICAO** sendo adicionadas à tabela **FUNCIONARIO** como opcionais.

Representação final: **FUNCIONARIO** (CODIGOFUNCIONARIO, NOME, CODIGONOTEBOOK, DESCRICAO).

O DER parcial mostra um relacionamento 1:1 com ambas as entidades de participação obrigatória (cardinalidades (1,1): (1,1)).



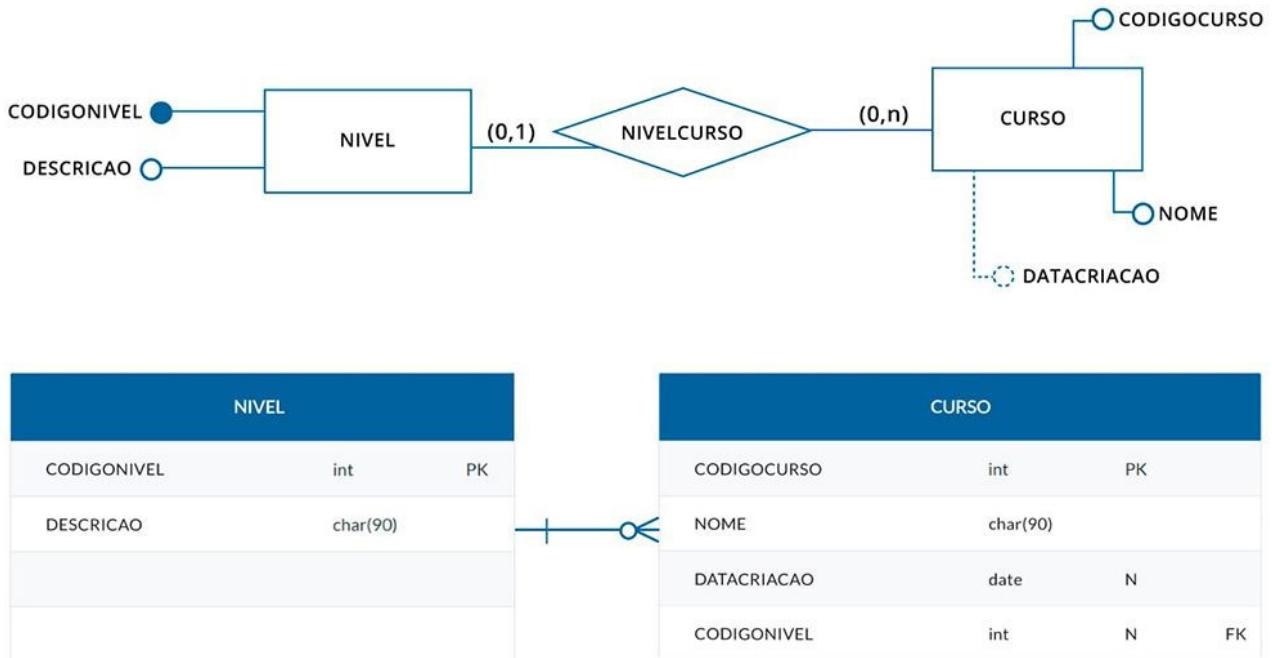
Modelo lógico gerado:

- **FUNCIONARIO**
 - CODIGOFUNCIONARIO (int, PK)
 - NOME (char(90))
 - CODIGONOTEBOOK (int, obrigatória)
 - DESCRICAO (char(90), obrigatória)

O modelo segue a regra de mapeamento conceitual-lógico para relacionamento 1:1, com ambas as entidades de participação obrigatória. Nesse caso, houve a fusão de tabelas, adicionando **CODIGONOTEBOOK** e **DESCRICAO** à tabela **FUNCIONARIO** como colunas obrigatórias.

A representação final da tabela **FUNCIONARIO**: (CODIGOFUNCIONARIO, NOME, CODIGONOTEBOOK, DESCRICAO).

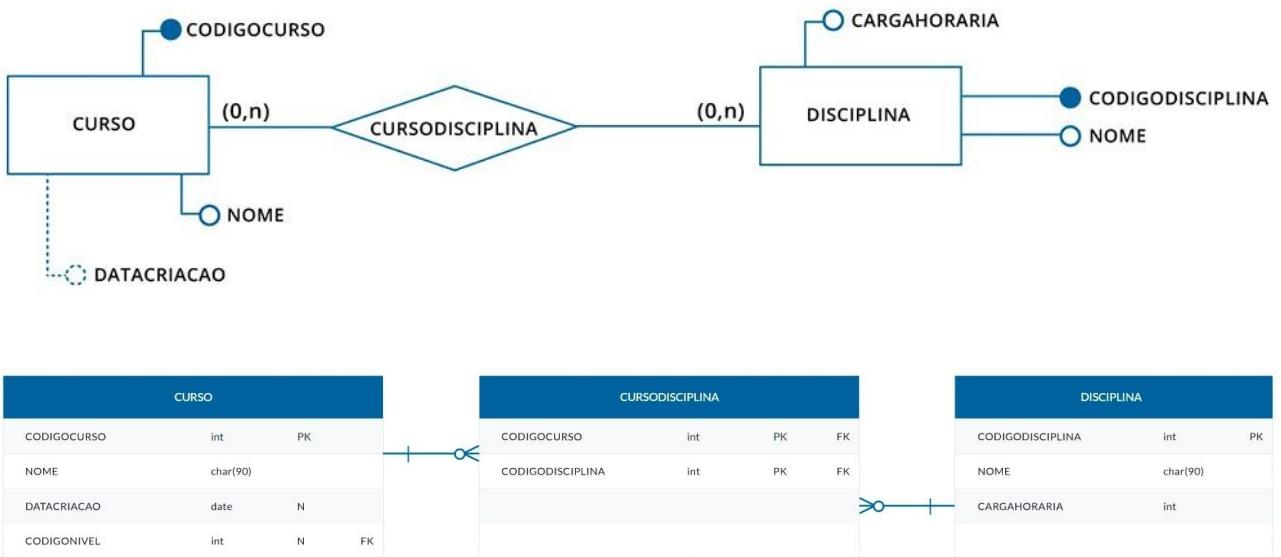
Exemplo de mapeamento de relacionamento 1:N com entidades **NIVEL** e **CURSO** em um DER parcial.



Para o relacionamento 1:N, foi adicionada a coluna **CODIGONIVEL** como chave estrangeira na tabela **CURSO**. A representação gerada foi:

- **NIVEL**: (CODIGONIVEL, DESCRICAO)
- **CURSO**: (CODIGOCURSO, NOME, DATACTRIACAO, CODIGONIVEL)
 - **CODIGONIVEL** REFERENCIA **NIVEL**

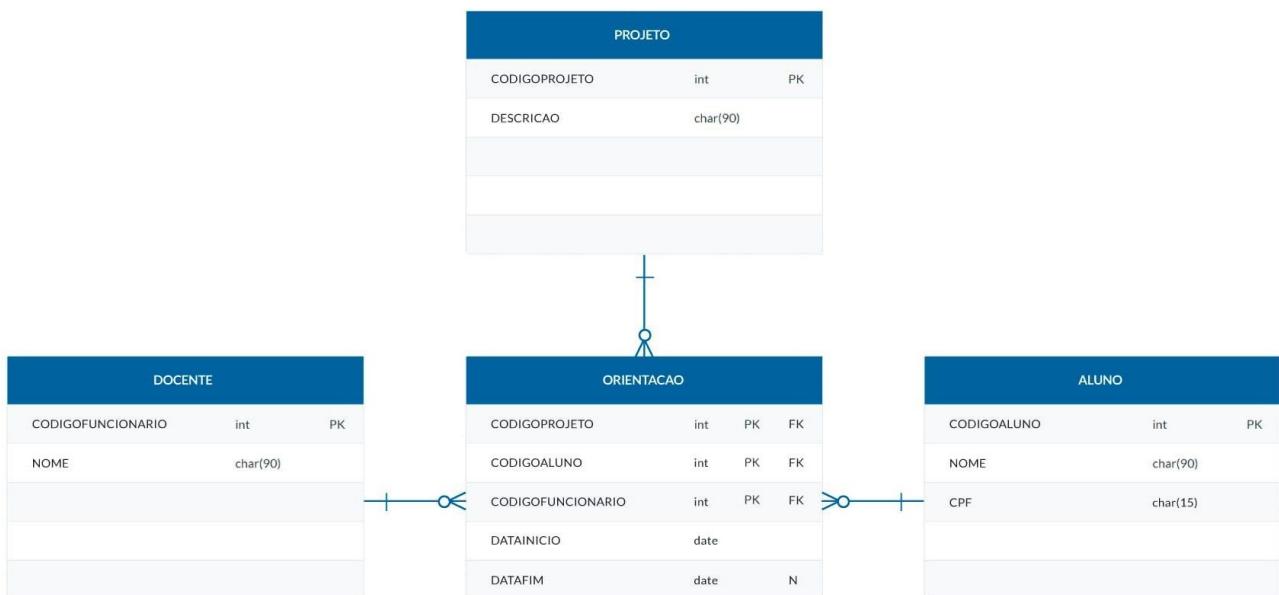
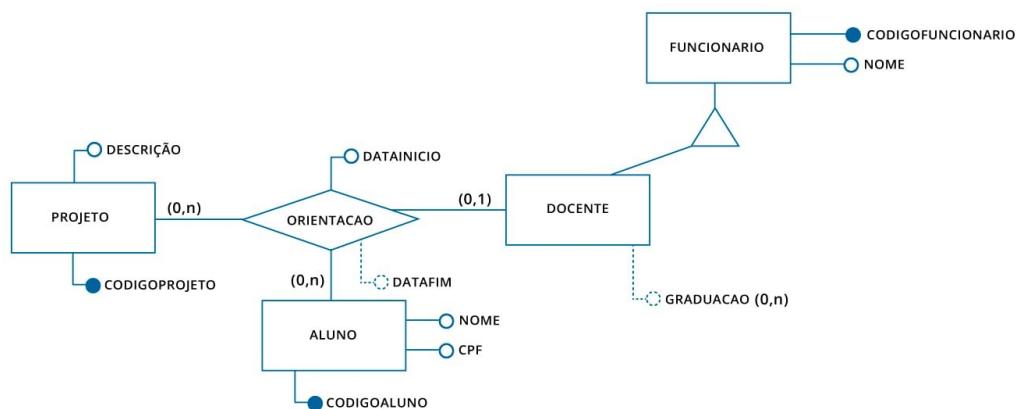
Exemplo de mapeamento de relacionamento N:N com entidades **CURSO** e **DISCIPLINA** em um DER parcial.



Para o relacionamento N:N, foi criada a tabela **CURSODISCIPLINA** com as chaves estrangeiras **CODIGOCURSO** e **CODIGODISCIPLINA**, sendo sua combinação a chave primária composta. A representação gerada foi:

- **CURSO**: (CODIGOCURSO, NOME, DATAACRIACAO, CODIGONIVEL)
 - **CODIGONIVEL** REFERENCIA **NIVEL**
- **DISCIPLINA**: (CODIGODISCIPLINA, NOME, CARGAGORARIA)
- **CURSODISCIPLINA**: (CODIGOCURSO, CODIGODISCIPLINA)
 - **CODIGOCURSO** REFERENCIA **CURSO**
 - **CODIGODISCIPLINA** REFERENCIA **DISCIPLINA**

Para entidades associativas, usa-se o mapeamento de relacionamentos N:N. Em autorrelacionamentos, as regras variam conforme a cardinalidade. Em relacionamentos ternários, é preciso avaliar as cardinalidades máximas. Exemplo: **PROJETO**, **DOCENTE** e **ALUNO** em um relacionamento ternário.



Para o relacionamento entre **PROJETO**, **DOCENTE** e **ALUNO**, foi criada a tabela **ORIENTACAO**, com três chaves estrangeiras (**CODIGOFUNCIONARIO**, **CODIGOALUNO**,

CODIGOPROJETO) e duas colunas adicionais (**DATAINICIO** e **DATAFIM**). A representação gerada foi:

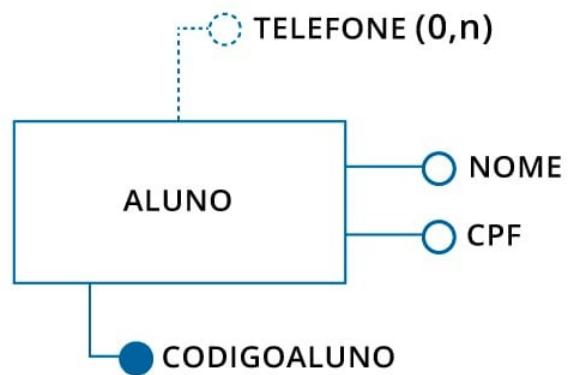
- **PROJETO**: (CODIGOPROJETO, DESCRICAO)
- **DOCENTE**: (CODIGOFUNCIONARIO, NOME)
- **ALUNO**: (CODOALUNO, NOME, CPF)
- **ORIENTACAO**: (CODIGOPROJETO, CODIGOALUNO, CODIGOFUNCIONARIO, DATAINICIO, DATAFIM)
 - **CODIGOPROJETO** REFERENCIA **PROJETO**
 - **CODIGOALUNO** REFERENCIA **ALUNO**
 - **CODIGOFUNCIONARIO** REFERENCIA **DOCENTE**

O mapeamento de atributos multivalorados envolve criar uma tabela para cada atributo com múltiplos valores, adicionando colunas para esses atributos. A tabela resultante possui uma chave estrangeira da tabela original e uma chave primária composta pela chave estrangeira e pelos atributos multivalorados.

Exemplo:

- **ALUNO**: (CODOALUNO, NOME, CPF)
- **FONEALUNO**: (CODOALUNO, NUMERO, TIPO)
 - **CODIGOALUNO** REFERENCIA **ALUNO**

Neste caso, a tabela **FONEALUNO** foi criada para o atributo **TELEFONE**, com **CODIGOALUNO** como chave estrangeira e chave primária composta por **CODIGOALUNO**, **NUMERO** e **TIPO** (tipo de telefone: residencial, comercial, móvel).



ALUNO			FONEALUNO			
CODIGOALUNO	int	PK	CODIGOALUNO	int	PK	FK
NOME	char(90)		NUMERO	char(15)	PK	
CPF	char(15)		TIPO	char(15)	PK	

Resumo: Mapeamento de Especialização/Generalização

O mapeamento de especialização/generalização modela hierarquias entre entidades em bancos de dados relacionais. Existem três soluções principais:

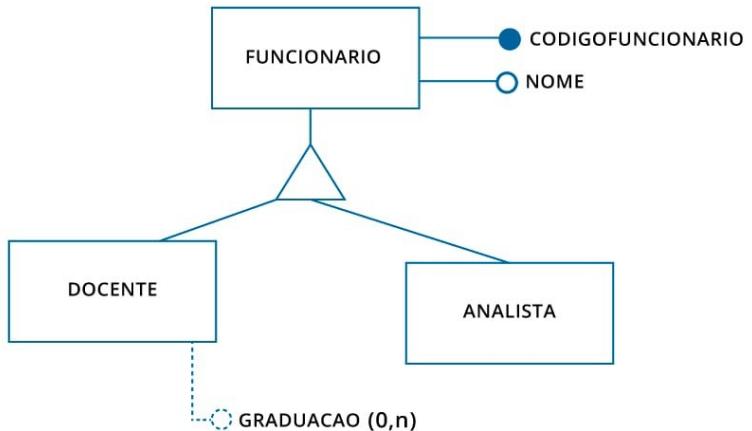
1. **Tabela única**: Criar uma única tabela com atributos das entidades genérica e especializadas, adicionando uma coluna **TIPO** para identificar a especialização.
2. **Tabelas separadas**: Criar uma tabela para a entidade genérica e outra para cada entidade especializada, com chave estrangeira referenciando a genérica.

3. **Subdivisão da entidade genérica:** Criar apenas tabelas especializadas, incorporando os atributos da entidade genérica. Se a hierarquia for parcial, uma tabela adicional pode ser necessária.

A terceira abordagem pode gerar redundância em hierarquias sobrepostas, exigindo controle adicional.

Resumo: Exemplos de Mapeamento de Especialização/Generalização

O mapeamento de especialização/generalização em bancos de dados relacionais pode ser feito de três formas, cada uma com vantagens e desvantagens:



1. Solução I - Tabela Única

- Cria-se uma única tabela (*FUNCIONARIO*) contendo atributos da entidade genérica (*CODIGOFUNCIONARIO*, *NOME*) e uma coluna adicional (*TIPO*) para identificar a especialização de cada funcionário.
- **Vantagens:** Estrutura simples e direta.
- **Desvantagens:** Pode gerar muitos valores nulos se as especializações tiverem atributos específicos.

Representação Textual:

FUNCIONARIO (CODIGOFUNCIONARIO, NOME, TIPO)

2. Solução II - Tabelas Separadas (Mais Usual)

- Cria-se uma tabela para a entidade genérica (*FUNCIONARIO*) e tabelas separadas para cada especialização (*DOCENTE* e *ANALISTA*).
- As tabelas especializadas possuem *CODIGOFUNCIONARIO* como chave primária e estrangeira, referenciando *FUNCIONARIO*.
- **Vantagens:** Mais flexível, facilita a adição de novas especializações sem modificar a estrutura existente.
- **Desvantagens:** Exige mais junções para consultar dados de diferentes especializações.

Representação Textual:

FUNCIONARIO (CODIGOFUNCIONARIO, NOME)
DOCENTE (CODIGOFUNCIONARIO) -- REFERENCIA FUNCIONARIO
ANALISTA (CODIGOFUNCIONARIO) -- REFERENCIA FUNCIONARIO

3. Solução III - Apenas Tabelas Especializadas

- Não há tabela para a entidade genérica. Cada especialização (*DOCENTE* e *ANALISTA*) possui sua própria tabela, contendo tanto os atributos da entidade genérica quanto os especializados.
- Se a especialização for **parcial**, cria-se uma tabela adicional (*OUTROSFUNCIONARIOS*) para armazenar funcionários sem especialização específica.
- **Vantagens:** Evita necessidade de junções para acessar atributos especializados.
- **Desvantagens:** Pode gerar **redundância de dados**, pois os atributos da entidade genérica são repetidos em cada tabela especializada.

Representação Textual:

DOCENTE (CODIGOFUNCIONARIO, NOME)

ANALISTA (CODIGOFUNCIONARIO, NOME)

OUTROSFUNCIONARIOS (CODIGOFUNCIONARIO, NOME) -- Se necessário

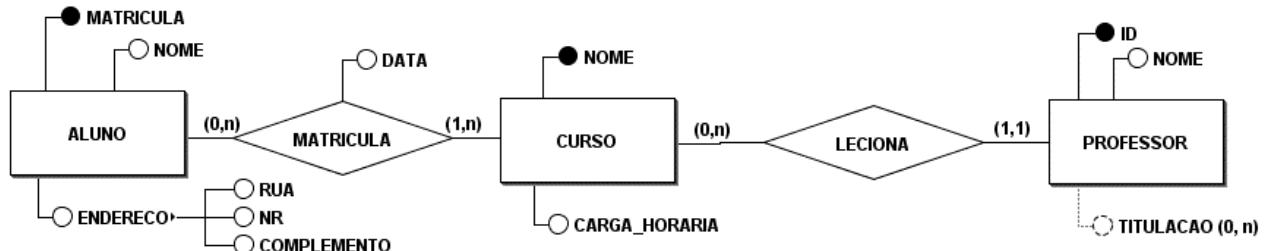
Considerações Finais

- **A Solução II é a mais usada** devido à sua flexibilidade para novas especializações.
- **A Solução I pode causar valores nulos**, tornando-se ineficiente se as especializações tiverem atributos distintos.
- **A Solução III pode gerar redundância**, exigindo um controle rigoroso para evitar inconsistências na chave primária.

Ao escolher a abordagem, deve-se considerar a necessidade de futuras especializações, a frequência de consultas e a eficiência na manipulação dos dados.

Resumo: Estudo de Caso de Mapeamento Conceitual-Lógico

O estudo de caso envolve a modelagem lógica da **Empresa de Treinamento**, partindo da modelagem conceitual representada em um **DER (Diagrama Entidade-Relacionamento)**. O processo segue um roteiro estruturado para transformar o modelo conceitual em um **modelo relacional**.



Passos para o Mapeamento Conceitual-Lógico:

1. **Criar tabelas** para cada entidade do modelo conceitual.
2. **Gerar colunas** para atributos simples e monovalorados.
3. **Marcar atributos únicos e obrigatórios** como identificadores.
4. **Definir as chaves primárias** das tabelas criadas.

5. **Decompor atributos compostos** em colunas separadas dentro da mesma tabela.
6. **Criar tabelas para atributos multivalorados**, incluindo chave estrangeira para a entidade de origem.
7. **Tratar relacionamentos N:N** criando uma nova tabela que contenha as chaves das entidades envolvidas.
8. **Tratar relacionamentos 1:1 e 1:N**, adicionando a chave estrangeira do lado 1 para o lado N (ou no outro lado, no caso de 1:1).
9. **Definir tipos de colunas** de acordo com os dados armazenados.
10. **Aplicar regras de generalização/especialização**, escolhendo entre as soluções possíveis (tabela única, tabelas separadas ou apenas tabelas especializadas).

Esse processo assegura a conversão correta do modelo conceitual para um banco de dados relacional eficiente, garantindo integridade e coerência na estrutura.

Resumo: Consultas e Transações em Banco de Dados

Este núcleo aborda **consultas e transações**, fundamentais para o funcionamento dos bancos de dados, destacando definições, características, benefícios e impacto no desempenho e confiabilidade do sistema.

Consultas

As consultas em bancos de dados são essenciais para recuperação, inclusão, exclusão e atualização de dados, sendo implementadas via **SQL (Structured Query Language)**. Um exemplo clássico de consulta para recuperar professoras na tabela DOCENTE seria:

```
SELECT CODIGODOCENTE, NOME
FROM DOCENTE
WHERE SEXO = 'F';
```

Aqui, o **SELECT** especifica as colunas retornadas, **FROM** indica a tabela e **WHERE** define o filtro aplicado à consulta. O desempenho do banco pode ser otimizado considerando as consultas e transações realizadas pela aplicação.

Transações

Uma **transação** é um conjunto de operações que devem ser tratadas como uma unidade lógica de trabalho. Se alguma falha ocorrer, todas as operações são canceladas, garantindo **consistência** no banco de dados.

Exemplo: No processo de **inscrição em disciplinas**, o aluno seleciona matérias antes de confirmar a inscrição. Esse processo deve ser tratado como uma única transação **atômica**—ou todas as operações são concluídas, ou nenhuma é realizada.

Conclusão

Entender **consultas e transações** permite projetar bancos de dados eficientes e confiáveis, assegurando integridade e otimização na manipulação dos dados.

Resumo: Indexação e Consultas com Múltiplas Tabelas

Este tema aborda **indexação** e **consultas com múltiplas tabelas**, essenciais para a recuperação eficiente de dados em bancos relacionais. A indexação melhora o desempenho das consultas, enquanto as consultas envolvendo várias tabelas, especialmente em sistemas normalizados, exigem o uso de **joins** para relacionar dados.

Consultas com Múltiplas Tabelas

A maioria das consultas em bancos de dados envolve múltiplas tabelas, como no exemplo abaixo:

Estrutura das tabelas

- **MUNICIPIO** (CO_MUNICIPIO, NOME)
- **DM_DOCENTE_2** (CO_DOCENTEIES, COIES, NOIES,
CO_MUNICIPIO_NASCIMENTO)
 - **CO_MUNICIPIO_NASCIMENTO** → Chave estrangeira referenciando **MUNICIPIO.CO_MUNICIPIO**

Objetivo: recuperar o código do docente e o nome do município de nascimento.

Consulta SQL

```
SELECT CO_DOCENTEIES, NOME  
FROM DM_DOCENTE_2, MUNICIPIO  
WHERE MUNICIPIO.CO_MUNICIPIO = DM_DOCENTE_2.CO_MUNICIPIO_NASCIMENTO;
```

- **SELECT** especifica as colunas de saída.
- **FROM** indica as tabelas envolvidas.
- **WHERE** estabelece a relação entre chave primária (**MUNICIPIO.CO_MUNICIPIO**) e chave estrangeira (**DM_DOCENTE_2.CO_MUNICIPIO_NASCIMENTO**).

Impacto no Desempenho

O SGBD cria uma **tabela temporária** combinando cada linha das tabelas envolvidas. Com 367.980 registros em **DM_DOCENTE_2** e 5.570 em **MUNICIPIO**, isso resultaria em mais de **2 bilhões de combinações**, tornando o processamento custoso.

Conclusão

Se esse tipo de consulta for frequente, pode causar lentidão no sistema. O uso adequado de **índices** e técnicas de otimização é essencial para evitar degradação de desempenho.

Resumo: Indexação em Banco de Dados

Indexação é uma técnica usada para otimizar consultas em bancos de dados, similar ao índice remissivo de um livro. Em bancos de dados, índices são estruturas auxiliares que tornam a recuperação de registros mais eficiente, especialmente em consultas com condições de busca, como igualdade na chave primária.

Funcionamento dos Índices

- Ao projetar uma tabela com chave primária, os registros são gravados sem uma ordem específica.
- O **SGBD** cria uma estrutura de índice para a chave primária para facilitar consultas. Isso melhora o desempenho da consulta.

Exemplo Prático

Considerando a tabela **DM_DOCENTE** com 367.980 registros, onde buscamos um docente com **CO_DOCENTEIES = 850516**, realizamos dois testes:

1. Consulta sem índice:

```
SELECT * FROM DM_DOCENTE WHERE CO_DOCENTEIES=850516;
```

- Tempo: **2,5 segundos**.

2. Consulta com índice (tabela **DM_DOCENTE_2** com chave primária):

- Tabela criada com os registros ordenados por **CO_DOCENTEIES**.
- Chave primária foi adicionada à tabela, criando um índice automaticamente.

```
SELECT * FROM DM_DOCENTE_2 WHERE CO_DOCENTEIES=850516;
```

- Tempo: **0,06 segundos**.

O resultado das consultas é o mesmo, mas a consulta com índice é significativamente mais rápida.

Conclusão

A **indexação** melhora a eficiência das consultas, especialmente quando há grandes volumes de dados.

Resumo: Projeto Físico em Bancos de Dados Relacionais

O **projeto físico** de um banco de dados envolve a criação da estrutura física, utilizando o **SGBD** e ferramentas **CASE** para gerar códigos SQL. O processo começa após o levantamento de requisitos, e abrange o projeto conceitual, lógico e físico.

Fatores que Influenciam o Projeto Físico

1. Consultas e Transações:

Para otimizar o desempenho, é necessário planejar as consultas e transações do banco. As informações essenciais para isso incluem:

- Tabelas acessadas
- Colunas usadas em condições de seleção
- Tipo de condição: intervalo, igualdade, desigualdade
- Colunas de junção
- Colunas nos resultados da consulta

2. Indexação:

Colunas que são frequentemente usadas em condições de seleção ou junção (como as mencionadas acima) devem ser indexadas.

Exemplos de Consultas

- **Consulta 1 (Seleção):**

A consulta para buscar cursos de **Medicina** ou **Nutrição** envolve a coluna **NOME**, que seria uma boa candidata a índice.

```
SELECT * FROM CURSO WHERE NOME='Medicina' OR NOME='Nutrição';
```

- **Consulta 2 (Junção):**

A consulta que recupera o nome do curso e nível utiliza uma junção entre as tabelas **CURSO** e **NIVEL**, sendo a coluna **CODIGONIVEL** uma boa candidata a índice.

```
SELECT NOME, DESCRIÇÃO FROM CURSO JOIN NIVEL ON CURSO.CODIGONIVEL=NIVEL.CODIGONIVEL;
```

- **Consulta 3 (Exclusão):**

A exclusão de cursos com a string “Engenharia” na coluna **NOME** faz com que essa coluna também seja uma boa candidata para índice.

```
DELETE FROM CURSO WHERE NOME LIKE '%ENGENHARIA%';
```

Considerações sobre Indexação

- **Boas candidatas para indexação:**

- Colunas em condições de seleção ou junção.

- **Evitar índices em colunas de atualização:**

- Colunas usadas em operações de exclusão ou atualização não devem ser indexadas.

O **projeto físico** envolve otimizar o acesso aos dados e ajustar o banco de dados para um desempenho eficiente, considerando consultas e transações esperadas.

Resumo: Frequência de Chamadas de Consultas e Transações Esperada

Identificar a frequência de consultas e transações esperadas é essencial para otimizar o desempenho do banco de dados. O **Princípio de Pareto (80/20)** sugere que 80% do processamento vem de 20% das consultas e transações. Portanto, é mais eficiente focar nas 20% mais relevantes.

Restrições de Tempo em Consultas e Transações

Consultas e transações podem ter **restrições de tempo** rigorosas. Por exemplo, uma transação de compras pode ter que ser concluída em até 7 segundos em 90% dos casos, e nunca ultrapassar 15 segundos. Essas restrições impactam as **colunas a serem indexadas**.

Frequência de Operações de Atualização

Evite criar **índices em colunas de tabelas frequentemente atualizadas**, pois a atualização desses índices pode gerar lentidão. Por exemplo, se uma tabela tem 5 colunas indexadas, cada inserção exige atualização desses índices.

Restrições de Exclusividade em Colunas

Crie índices para colunas com **restrições de unicidade**, como em operações de inserção. O SGBD usará o índice para validar a exclusividade, rejeitando inserções com valores duplicados.

Resumo: Prática de Instalação de SGBD e Criação de Banco de Dados

Neste exercício, o objetivo é praticar a instalação do **SGBD PostgreSQL** e a criação de um banco de dados.

Roteiro da Prática:

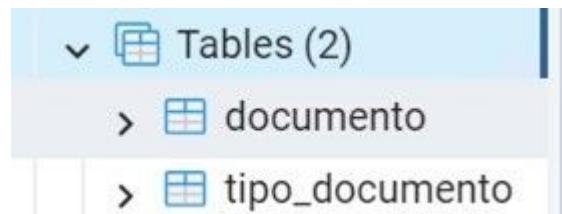
1. Gere o **script** do banco de dados de documentos.
2. Acesse o arquivo **DOCUMENTOLOGICO.brM3** no **brmodelo3** e gere o **modelo físico**.
3. Salve o script, disponível para download como **criadoc.sql**.

Passos para Instalar o PostgreSQL:

1. Acesse o site oficial do PostgreSQL e baixe a versão compatível com seu sistema operacional.
2. Execute o instalador, seguindo as instruções para configurar o PostgreSQL, criando uma senha para o superusuário e definindo as opções necessárias (porta padrão 5432, diretórios de dados e configuração).
3. Complete a instalação e, se necessário, reinicie o sistema.
4. Instale a ferramenta **PgAdmin4**.

Criação do Banco de Dados:

1. Abra o **PgAdmin4** e conecte-se ao servidor.
2. Crie o banco de dados **documentos**.
3. Abra a janela de consulta, carregue o **script** e execute-o.
4. Valide a criação das tabelas na aba **Tabelas** do banco de dados **documentos**.



Resumo: Prática de Consultas com Múltiplas Tabelas e Indexação

Neste exercício, vamos complementar o banco de documentos com a criação de duas tabelas e explorar o uso de índices para otimizar as consultas SQL.

Roteiro da Prática:

1. **Complementar o banco de documentos:**
 - Abra o **PGAdmin4**, conecte-se ao servidor e selecione o banco de dados **documentos**.
 - Abra a janela de consulta, carregue e execute o script.
 - Valide a criação das tabelas na aba **Tabelas**.
2. **Uso de índices:**
 - Rode o script **criapopdespacho.sql**.

- Realize consultas na tabela **despacho**, buscando por um documento específico.
- Crie um índice na coluna **documento** da tabela **despacho**.
- Realize as consultas novamente e compare o tempo de resposta antes e depois da criação do índice.

▼	Tables (4)
>	despacho
>	documento
>	tipo_despacho
>	tipo_documento

Resumo: Desnormalização para Ganhar Desempenho

A desnormalização é uma técnica usada para otimizar o desempenho de consultas em bancos de dados relacionais, sacrificando as vantagens de um modelo normalizado para reduzir o custo de acesso a múltiplas tabelas. Isso é comum em sistemas que geram relatórios.

Princípios da Desnormalização:

- Em um banco normalizado até a 3FN, a redundância de dados é minimizada, mas acessar várias tabelas para uma consulta gera custo adicional.
- Desnormalizar envolve criar uma estrutura com uma tabela única, sacrificando a normalização (como no exemplo onde a dependência funcional parcial viola a 2FN).
- O processo pode melhorar o desempenho de consultas, mas introduz redundância e exige atualizações adicionais para manter a consistência.

Considerações Importantes:

A desnormalização deve ser planejada, pois, embora melhore o desempenho, ela também aumenta a complexidade de manutenção.

Introdução ao PostgreSQL e SQL

Este tema aborda o PostgreSQL, um SGBD de código aberto desenvolvido em C, disponível em diversos sistemas operacionais como Linux, Windows, e OS X. O foco é aprender a instalar o PostgreSQL em Linux e Windows, explorar comandos SQL, especialmente DDL, e entender os comandos CRUD (criação, consulta, atualização e remoção de dados).

Além disso, será explicado o conceito de transações, onde um conjunto de comandos deve ser executado como uma unidade atômica, garantindo que todas as operações sejam realizadas ou nenhuma seja.

Breve histórico do PostgreSQL

O PostgreSQL originou-se do projeto POSTGRES, que derivou do INGRES da Universidade da Califórnia em Berkeley. O desenvolvimento começou em 1986 e tornou-se operacional em 1987, com a primeira versão pública lançada em 1989. O Postgres95 foi lançado em 1995, adotando SQL como interface padrão. Em 1996, foi renomeado para PostgreSQL, a partir da versão 6, e evoluiu para um dos principais SGBDs de código aberto, com versões para Windows, Mac OS e Linux.

Arquitetura do PostgreSQL

O PostgreSQL adota o modelo cliente-servidor. O **servidor** gerencia arquivos de banco de dados, conexões e executa comandos. O **cliente** solicita acesso, envia comandos de manipulação e consultas. A comunicação entre eles, frequentemente via TCP/IP, pode ocorrer em máquinas diferentes, localmente ou remotamente. O PostgreSQL suporta múltiplas conexões simultâneas, iniciando um processo para cada nova conexão.

Os modelos **conceitual, lógico e físico** representam diferentes níveis de abstração no design de bancos de dados:

1. Modelo Conceitual

- Representa a estrutura **geral** do banco, sem detalhes técnicos.
- Foca nos **dados e suas relações**, geralmente usando diagramas (como **DER – Diagrama Entidade-Relacionamento**).
- Exemplo: Cliente (CPF, Nome, Endereço) → realiza → Pedido (ID, Data, Valor).

2. Modelo Lógico

- Traduz o modelo conceitual para um formato mais estruturado, seguindo um **modelo de dados específico** (relacional, orientado a objetos, etc.).
- Define **tabelas, atributos, chaves primárias e estrangeiras**, sem depender de um SGBD específico.
- Exemplo (modelo relacional):
 - **Cliente** (CPF *PK*, Nome, Endereço)
 - **Pedido** (ID *PK*, Data, Valor, CPF *FK*).

3. Modelo Físico

- Implementação detalhada no **SGBD escolhido**.
- Define **tipos de dados, índices, estrutura de armazenamento e otimizações**.
- Exemplo (SQL para um banco relacional):

```
CREATE TABLE Cliente (
    CPF CHAR(11) PRIMARY KEY,
    Nome VARCHAR(100),
    Endereco TEXT
);

CREATE TABLE Pedido (
    ID INT PRIMARY KEY,
    Data DATE,
    Valor DECIMAL(10, 2),
    CPF CHAR(11),
    FOREIGN KEY (CPF) REFERENCES Cliente(CPF)
);
```

Cada modelo refina o anterior até a implementação final no banco de dados.

A diferença entre **DML (Data Manipulation Language)** e **DDL (Data Definition Language)** está no objetivo de cada conjunto de comandos dentro de um banco de dados:

1. DML (Linguagem de Manipulação de Dados)

- Usada para **manipular os dados armazenados** no banco.
- **Não altera a estrutura das tabelas**, apenas insere, atualiza, exclui ou recupera informações.
- Comandos principais:
 - **SELECT** – Consulta dados.
 - **INSERT** – Insere novos registros.
 - **UPDATE** – Modifica registros existentes.
 - **DELETE** – Remove registros.

2. DDL (Linguagem de Definição de Dados)

- Usada para **definir ou modificar a estrutura do banco de dados**.
- Atua na criação, modificação e remoção de **tabelas, índices e esquemas**.
- Comandos principais:
 - **CREATE** – Cria tabelas, bancos de dados, índices, etc.
 - **ALTER** – Modifica a estrutura de tabelas (adicionar/remover colunas, restrições, etc.).
 - **DROP** – Remove tabelas, bancos de dados ou objetos.
 - **TRUNCATE** – Remove todos os dados de uma tabela, sem afetar sua estrutura.

Resumo

Categoria	Objetivo	Exemplos de Comandos
DML	Manipulação de dados	SELECT, INSERT, UPDATE, DELETE
DDL	Definição da estrutura	CREATE, ALTER, DROP, TRUNCATE

O **DML trabalha com os dados dentro das tabelas**, enquanto o **DDL modifica a estrutura das tabelas e do banco de dados**.

Criando um Banco de Dados no PostgreSQL

Para criar um banco de dados no PostgreSQL, utilize:

```
CREATE DATABASE BDESTUDO;
```

Para removê-lo, use:

```
DROP DATABASE BDESTUDO;
```

Todo banco criado possui um schema padrão chamado **public**, onde as tabelas são armazenadas por padrão. Para usar um schema diferente, é necessário criá-lo com:

```
CREATE SCHEMA esquema;
```

Tabelas dentro desse schema devem ser referenciadas pelo nome completo: esquema.tabela.

Criando Tabelas no PostgreSQL

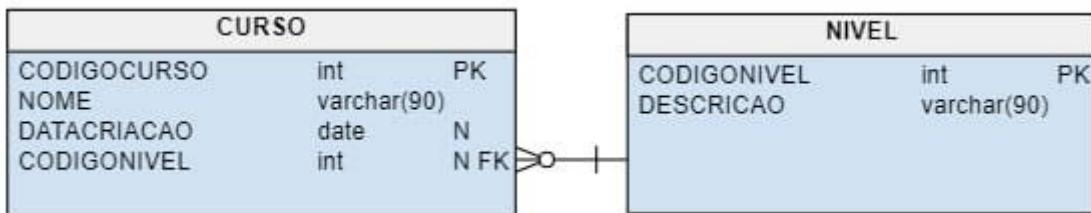
No PostgreSQL, as tabelas são criadas com o comando `CREATE TABLE`. A sintaxe básica é:

```
CREATE TABLE NOMETABELA (
    COLUNA1 TIPODEDADOS [RESTRIÇÃO],
    COLUNAN TIPODEDADOS [RESTRIÇÃO],
    PRIMARY KEY (COLUNA),
    FOREIGN KEY (COLUNA) REFERENCES OUTRATABELEA (COLUNA),
    CONSTRAINT RESTRIÇÃO
);
```

Elementos da Sintaxe:

- **NOMETABELA**: Nome da tabela.
- **COLUNA1, COLUNAN**: Nomes das colunas.
- **TIPODEDADOS**: Tipo de dado da coluna.
- **RESTRIÇÃO**: Propriedades como obrigatoriedade.
- **PRIMARY KEY**: Define a chave primária.
- **FOREIGN KEY**: Define a chave estrangeira.
- **CONSTRAINT**: Define restrições adicionais.

A documentação completa do comando está disponível no site oficial do PostgreSQL.



Tipos de Dados no PostgreSQL

Cada coluna em uma tabela deve ter um tipo de dado. No PostgreSQL, os principais tipos são:

- **bigint**: Inteiros de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807.
- **int (integer)**: Inteiros de -2.147.483.648 a 2.147.483.647.
- **smallint**: Inteiros de -32.768 a 32.767.
- **decimal, numeric**: Definem precisão decimal.
- **real**: Até 6 casas decimais.
- **double**: Até 15 casas decimais.
- **money**: Valores monetários de -92.233.720.368.547.758,08 a 92.233.720.368.547.758,07.
- **serial**: Gera valores inteiros sequenciais de 1 a 2.147.483.647.
- **char(comprimento)**: Cadeia de caracteres de tamanho fixo, preenchida com espaços se necessário.
- **varchar(comprimento)**: Cadeia de caracteres de tamanho variável, sem preenchimento de espaços extras.

- **date**: Datas no formato AAAA-MM-DD.
- **time**: Representa horários entre 00:00:00 e 24:00:00.

Mais detalhes podem ser encontrados na documentação oficial do PostgreSQL.

Exemplo de Criação de Tabelas

O código SQL cria as tabelas **NIVEL** e **CURSO** no banco de dados **bdestudo**.

- **Tabela NIVEL** (linhas 2-6): Possui duas colunas obrigatórias.
- **Tabela CURSO** (linhas 8-15): As colunas **DATAACRIACAO** e **CODIGONIVEL** são opcionais. **CODIGONIVEL** permite associar um curso a um nível posteriormente.
- **Schema**: Como não foi especificado, as tabelas são armazenadas no schema padrão **public**.
- **Restrição UNIQUE**: A coluna **NOME** da tabela **NIVEL** foi declarada **UNIQUE**, garantindo que valores duplicados não sejam permitidos.

```

1 -- Comando para criar a tabela nivel
2 CREATE TABLE NIVEL (
3   CODIGONIVEL int NOT NULL,
4   DESCRICAO varchar(90) NOT NULL,
5   CONSTRAINT CHAVEPNIVEL PRIMARY KEY (CODIGONIVEL)
6 );
7 -- Comando para criar a tabela curso
8 CREATE TABLE CURSO (
9   CODIGOCURSO int NOT NULL,
10  NOME varchar(90) NOT NULL UNIQUE,
11  DATAACRIACAO date NULL,
12  CODIGONIVEL int NULL,
13  CONSTRAINT CHAVEPCURSO PRIMARY KEY (CODIGOCURSO),
14  FOREIGN KEY (CODIGONIVEL) REFERENCES NIVEL (CODIGONIVEL)
15 );

```

Gerenciamento de Scripts na Prática

O script SQL apresentado tem 15 linhas e serve como exemplo didático com apenas duas tabelas. Projetos reais geralmente possuem dezenas de tabelas.

Embora seja essencial conhecer comandos **DDL**, a prática ocorre com ferramentas que automatizam a criação e administração de bancos de dados. Elas permitem definir tabelas, relacionamentos e restrições visualmente, além de gerar código **DDL** completo ou parcial. Exemplos: **DBeaver**, **Vertabelo**, **SQL Power Architect**, **Toad for SQL** e **Erwin**.

SGBD PostgreSQL nos Bastidores

Ao executar **CREATE DATABASE**, o PostgreSQL realiza diversas etapas internas. Em uma instalação padrão no Windows, o servidor cria uma pasta identificada por um número (**OID**) dentro do diretório:

C:\Program Files\PostgreSQL\12\data\base

OS (C) > Arquivos de Programas > PostgreSQL > 12 > data > base			
	Nome	Data de modificação	Tipo
+	1	26/06/2020 17:28	Pasta de arquivos
+	13317	04/06/2020 14:34	Pasta de arquivos
+	13318	26/06/2020 06:04	Pasta de arquivos
+	24600	28/06/2020 13:14	Pasta de arquivos
+	41015	30/06/2020 10:42	Pasta de arquivos
+	41016	30/06/2020 10:52	Pasta de arquivos

Após a execução do comando `CREATE DATABASE TESTEBANCO;` foi criada a pasta 41017, conforme imagem a seguir:

OS (C:) > Arquivos de Programas > PostgreSQL > 12 > data > base			
	Nome	Data de modificação	Tipo
1	1	26/06/2020 17:28	Pasta de arquivos
2	13317	04/06/2020 14:34	Pasta de arquivos
3	13318	26/06/2020 06:04	Pasta de arquivos
4	24600	28/06/2020 13:14	Pasta de arquivos
5	41015	30/06/2020 10:42	Pasta de arquivos
6	41016	30/06/2020 10:52	Pasta de arquivos
7	41017	30/06/2020 12:32	Pasta de arquivos

Catálogo do PostgreSQL

O PostgreSQL armazena informações sobre todos os bancos de dados em um **catálogo** especial, cujas tabelas começam com **PG_**. A tabela **PG_DATABASE** contém detalhes sobre os databases do servidor. Para identificar o nome de um banco a partir do **OID**, use:

```
SELECT OID, DATNAME FROM PG_DATABASE;
```

Os resultados podem variar conforme as operações realizadas no SGBD.

	oid	datname
1	1	template1
2	13.317	template0
3	13.318	postgres
4	24.600	bdestudo
5	41.015	bdtestepgadmin
6	41.016	bdtestepsq
7	41.017	testebanco

Alteração de Tabela no PostgreSQL

Quando for necessário alterar a estrutura de uma tabela, utiliza-se o comando **ALTER TABLE**.

Adicionar uma coluna

A sintaxe básica para adicionar uma coluna a uma tabela existente é:

```
ALTER TABLE <NOMETABELA> ADD <COLUNA> <TIPODEDADOS>;
```

- <NOMETABELA>: Nome da tabela.
- <COLUNA>: Nome da coluna a ser adicionada.
- <TIPODEDADOS>: Tipo de dados da coluna.

Exemplo:

```
ALTER TABLE CURSO ADD DTRECONH DATE;
```

Por padrão, a nova coluna será opcional (**NULL**).

Remover uma coluna

Para remover uma coluna de uma tabela, usa-se:

```
ALTER TABLE <NOMETABELA> DROP <COLUNA>;
```

Exemplo:

```
ALTER TABLE CURSO DROP DTRECONH;
```

Esses comandos permitem modificar a estrutura das tabelas conforme a necessidade de modelagem dos dados. A sintaxe completa pode ser consultada na documentação oficial do PostgreSQL.

Remoção de Tabela no PostgreSQL

Para remover uma tabela, usa-se o comando **DROP TABLE** com a seguinte sintaxe:

```
DROP TABLE <NOMETABELA>;
```

Exemplo para remover a tabela **CURSO**:

```
DROP TABLE CURSO;
```

Além disso, é importante tomar cuidados ao manipular tabelas relacionadas.

Criação e Alteração de Tabelas Relacionadas

As tabelas **NIVEL** e **CURSO** podem ser criadas sem relacionamento e, depois, a tabela **CURSO** pode ser alterada para adicionar a chave estrangeira. O script SQL seria modificado da seguinte forma:

1. Criar as tabelas sem chave estrangeira.
2. Usar o comando **ALTER TABLE** para adicionar a restrição de chave estrangeira, criando o relacionamento entre **CURSO** e **NIVEL**.

```
1 -- comando para criar a tabela nivel
2 CREATE TABLE NIVEL (
3     CODIGONIVEL int NOT NULL,
4     DESCRICAO varchar(90) NOT NULL,
5     CONSTRAINT CHAVEPNIVEL PRIMARY KEY (CODIGONIVEL));
6 -- comando para criar a tabela curso
7 CREATE TABLE CURSO (
8     CODIGOCURSO int NOT NULL,
9     NOME varchar(90) NOT NULL UNIQUE,
10    DATAACRIACAO date NULL,
11    CODIGONIVEL int NULL,
12    CONSTRAINT CHAVEPCURSO PRIMARY KEY (CODIGOCURSO));
13 -- comando para alterar a tabela curso, adicionando chave estrangeira
14 ALTER TABLE CURSO ADD FOREIGN KEY (CODIGONIVEL) REFERENCES NIVEL;
```

Cuidados ao Manipular Tabelas Relacionadas

Em bancos de dados relacionais, o relacionamento entre tabelas é feito por chave estrangeira, criando dependências entre elas. Quando uma tabela possui uma chave estrangeira, ela aponta para uma chave primária em outra tabela, estabelecendo uma relação.

Ao manipular tabelas relacionadas, o SGBD garante a integridade dos dados e pode bloquear ações que resultem em inconsistências, mesmo que o comando SQL esteja correto. Por exemplo, ao tentar remover a tabela **NIVEL**, o SGBD impedirá a ação se outra tabela, como **CURSO**, depender dela.

Remoção de Tabelas com Dependências

Se a tabela **NIVEL** for removida sem cautela, a tabela **CURSO** ficará inconsistente devido à chave estrangeira **CODIGONIVEL**. Para evitar isso, é preciso remover as dependências antes de excluir a tabela.

Caso queira remover a tabela **NIVEL** com dependências, o comando **DROP TABLE NIVEL CASCADE** pode ser usado. Isso remove automaticamente as dependências e a tabela **NIVEL**, ajustando as tabelas relacionadas, como **CURSO**.

Este módulo abordou os comandos SQL básicos para criar, alterar e remover tabelas no PostgreSQL, além dos tipos de dados comuns.

Manipulação de Linhas nas Tabelas

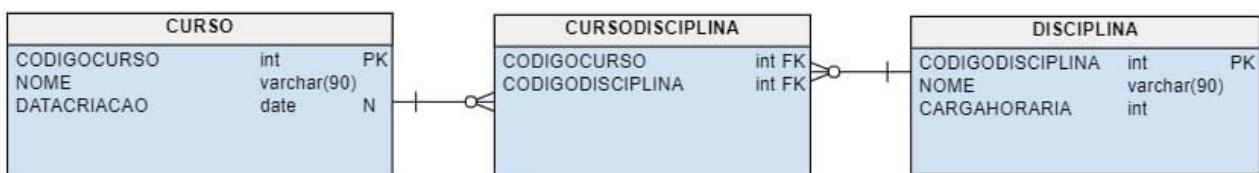
A manipulação de dados envolve inserção, atualização e remoção, representadas pelas operações CRUD (Create, Read, Update, Delete). No SQL, os comandos correspondentes são:

- **Create:** INSERT
- **Read:** SELECT
- **Update:** UPDATE
- **Delete:** DELETE

Esses comandos fazem parte da DML (Data Manipulation Language), usados para modificar dados nas tabelas.

Modelo para os Exemplos

O modelo no banco de dados inclui as tabelas **CURSO**, **CURSODISCIPLINA** e **DISCIPLINA**, usado para gerenciar cursos, disciplinas e seus relacionamentos. Cada linha em **CURSODISCIPLINA** associa um curso a uma disciplina. Recomenda-se criar as tabelas e inserir dados usando um script, conectado ao PostgreSQL e ao banco **bdestudo**.



```
1 CREATE TABLE CURSO (
2   CODIGOCURSO int NOT NULL,
3   NOME varchar(90) NOT NULL,
4   DATAACRIACAO date NULL,
5   CONSTRAINT CURSO_pk PRIMARY KEY (CODIGOCURSO));
6 CREATE TABLE DISCIPLINA (
7   CODIGODISCIPLINA int NOT NULL,
8   NOME varchar(90) NOT NULL,
9   CARGAHORARIA int NOT NULL,
10  CONSTRAINT DISCIPLINA_pk PRIMARY KEY (CODIGODISCIPLINA));
11 CREATE TABLE CURSODISCIPLINA (
12   CODIGOCURSO int NOT NULL,
13   CODIGODISCIPLINA int NOT NULL,
14   CONSTRAINT CURSODISCIPLINA_pk PRIMARY KEY (CODIGOCURSO,CODIGODISCIPLINA),
15   CONSTRAINT CURSODISCIPLINA_CURSO FOREIGN KEY (CODIGOCURSO) REFERENCES CURSO (CODIGOCURSO) ON DELETE CASCADE ,
16   CONSTRAINT CURSODISCIPLINA_DISCIPLINA FOREIGN KEY (CODIGODISCIPLINA) REFERENCES DISCIPLINA (CODIGODISCIPLINA) );
```

Inserção de Linhas em Tabela

A inserção de dados em tabelas é feita com o comando **INSERT**. Sua sintaxe é:

```
INSERT INTO <NOMETABELA> (COLUNA1, COLUNA2,..., COLUNAn) VALUES (VALOR1, VALOR2,...,  
VALORn);
```

Cada valor deve corresponder ao tipo da coluna. Exemplo de inserção de cursos:

```
INSERT INTO CURSO (CODIGOCURSO, NOME, DATAACRIACAO) VALUES (1, 'Sistemas de  
Informação', '19/06/1999');  
INSERT INTO CURSO (CODIGOCURSO, NOME, DATAACRIACAO) VALUES (2, 'Medicina',  
'10/05/1990');  
INSERT INTO CURSO (CODIGOCURSO, NOME, DATAACRIACAO) VALUES (3, 'Nutrição',  
'19/02/2012');  
INSERT INTO CURSO (CODIGOCURSO, NOME, DATAACRIACAO) VALUES (4, 'Pedagogia',  
'19/06/1999');
```

Valores inteiros não precisam de aspas, enquanto **char** e **date** precisam. O PostgreSQL converte o texto para o formato de data.

Inserção de Disciplinas e Associações

O comando **INSERT** é utilizado para cadastrar disciplinas:

```
INSERT INTO DISCIPLINA (CODIGODISCIPLINA, NOME, CARGAHORARIA) VALUES (1,  
'Leitura e Produção de Textos', 60);  
INSERT INTO DISCIPLINA (CODIGODISCIPLINA, NOME, CARGAHORARIA) VALUES (2, 'Redes  
de Computadores', 60);  
INSERT INTO DISCIPLINA (CODIGODISCIPLINA, NOME, CARGAHORARIA) VALUES (3,  
'Computação Gráfica', 40);  
INSERT INTO DISCIPLINA (CODIGODISCIPLINA, NOME, CARGAHORARIA) VALUES (4,  
'Anatomia', 60);
```

Em seguida, são registradas associações entre cursos e disciplinas na tabela **CURSODISCIPLINA**:

```
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (1, 1);  
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (1, 2);  
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (1, 3);  
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (2, 1);  
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (2, 3);  
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (3, 1);  
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (3, 3);
```

Se for inserido um valor de **CODIGODISCIPLINA** inexistente, como **30**:

```
INSERT INTO CURSODISCIPLINA(CODIGOCURSO, CODIGODISCIPLINA) VALUES (3, 30);
```

O SGBD retorna um erro, pois **30** não existe na tabela **DISCIPLINA**, garantindo a integridade dos dados.

Mecanismo de Chave Primária

O SGBD garante a integridade dos dados. Ao tentar inserir um valor duplicado na chave primária, como:

```
INSERT INTO DISCIPLINA (CODIGODISCIPLINA, NOME, CARGAHORARIA) VALUES (4,  
'Anatomia', 60);
```

O SGBD retorna um erro, pois o valor da chave primária já existe.

Além disso, ao tentar inserir um valor **NULL** em uma chave primária:

```
INSERT INTO DISCIPLINA (CODIGODISCIPLINA, NOME, CARGAHORARIA) VALUES (NULL, 'Biologia Celular e Molecular', 60);
```

O SGBD exibe um erro, pois a chave primária não pode ser **NULL**.

Atualização de Linhas em Tabela

A atualização de linhas é feita com o comando **UPDATE**:

```
UPDATE <NOMETABELA>
SET COLUNA1=VALOR1, COLUNA2=VALOR2, ..., COLUNAn=VALORn
WHERE <CONDIÇÃO>;
```

Exemplo 1: Para atualizar a carga horária da disciplina "Redes de Computadores":

```
UPDATE DISCIPLINA SET CARGAHORARIA=70 WHERE CODIGODISCIPLINA=2;
```

Exemplo 2: Alterar a carga horária para todas as disciplinas:

```
UPDATE DISCIPLINA SET CARGAHORARIA=CARGAHORARIA*1.2;
```

Neste caso, a cláusula **WHERE** não foi necessária, pois todas as linhas foram alteradas.

Atualização de Coluna Chave Primária

Alterar valores de uma coluna chave primária deve ser feito com cautela.

Exemplo 1: Alterando o valor de CODIGOCURSO:

```
UPDATE CURSO SET CODIGOCURSO=6 WHERE CODIGOCURSO=4;
```

Neste caso, o SGBD permite a alteração, pois não há vínculos na tabela **CURSODISCIPLINA**.

Exemplo 2: Tentando alterar CODIGOCURSO para 10:

```
UPDATE CURSO SET CODIGOCURSO=10 WHERE CODIGOCURSO=1;
```

O SGBD rejeita a alteração, pois o valor 1 está registrado em **CURSODISCIPLINA**, evitando dados inconsistentes.

Alteração de Chave Estrangeira com Atualização em Cascata

Usamos o comando **ALTER TABLE** para alterar a tabela **CURSODISCIPLINA**, removendo e recriando a chave estrangeira com **ON UPDATE CASCADE**. Isso permite que, ao atualizar o valor da chave primária, as referências na tabela **CURSODISCIPLINA** também sejam atualizadas automaticamente.

Comando para alteração:

```
ALTER TABLE CURSODISCIPLINA
DROP CONSTRAINT CURSODISCIPLINA_CURSO,
```

```
ADD CONSTRAINT CURSODISCIPLINA_CURSO
FOREIGN KEY (CODIGOCURSO) REFERENCES CURSO (CODIGOCURSO)
ON UPDATE CASCADE;
```

Após isso, podemos executar a atualização:

```
UPDATE CURSO SET CODIGOCURSO=10 WHERE CODIGOCURSO=1;
```

Resumo: Remoção de Linhas em Tabela

A remoção de linhas é feita com o comando **DELETE**:

```
DELETE FROM <NOMETABELA> WHERE <CONDIÇÃO>;
```

O comando especifica a tabela e uma condição, removendo as linhas que atendem a essa condição. Por exemplo, para apagar a disciplina de Anatomia:

```
DELETE FROM DISCIPLINA WHERE CODIGODISCIPLINA=4;
```

O SGBD remove a linha correspondente à condição.

Remoção de Linhas com Chave Estrangeira

Ao tentar remover a disciplina **Leitura e Produção de Textos** com o comando:

```
DELETE FROM DISCIPLINA WHERE CODIGODISCIPLINA=1;
```

o SGBD retornará um erro, pois o valor 1 está registrado na tabela **CURSODISCIPLINA**, o que causaria inconsistência. Para manter a integridade, utilizamos o comando **ALTER TABLE**:

```
ALTER TABLE CURSODISCIPLINA
DROP CONSTRAINT CURSODISCIPLINA_DISCIPLINA,
ADD CONSTRAINT CURSODISCIPLINA_DISCIPLINA
FOREIGN KEY (CODIGODISCIPLINA) REFERENCES DISCIPLINA (CODIGODISCIPLINA)
ON DELETE CASCADE;
```

Isso configura a remoção em cascata, permitindo que, ao executar o comando de remoção na tabela **DISCIPLINA**, as linhas correspondentes na **CURSODISCIPLINA** também sejam removidas automaticamente.

Eliminação de Todos os Registros

Para eliminar todos os registros do banco de dados, é necessário identificar tabelas independentes e aquelas com vínculos de chave estrangeira. A tabela **CURSODISCIPLINA** é a mais dependente, possuindo chaves estrangeiras, enquanto **CURSO** e **DISCIPLINA** são independentes. Os comandos para remover os registros são:

```
DELETE FROM CURSODISCIPLINA;
DELETE FROM CURSO;
DELETE FROM DISCIPLINA;
```

Primeiro, remove-se os registros de **CURSODISCIPLINA**, o que permite a remoção bem-sucedida das tabelas **CURSO** e **DISCIPLINA**. Este módulo abordou os comandos básicos de SQL para manipulação de dados no PostgreSQL.

Transações em Banco de Dados

Durante os estudos, aprendemos sobre comandos SQL, como criação de tabelas e manipulação de dados (inserções, atualizações e exclusões), onde um comando era enviado ao SGBD, que o executava e retornava um resultado. Em um ambiente de produção, o SGBD gerencia múltiplas requisições simultâneas, com diferentes usuários acessando recursos ao mesmo tempo. Por exemplo, em um sistema acadêmico, quando um aluno se inscreve em várias disciplinas, o SGBD executa várias inserções de dados, garantindo que a inscrição seja realizada corretamente e que não haja problemas como a falta de vagas. Isso implica que o SGBD precisa garantir a integridade dos dados ao processar várias operações ao mesmo tempo.

Transações em Banco de Dados

Em sistemas de banco de dados, uma transação é uma unidade de trabalho, iniciada com o comando *begin transaction* e finalizada com *end transaction*. Transações podem ser de leitura (não alteram o banco) ou leitura-gravação (alteram dados). Em SGBDs multiusuários, múltiplas transações podem ocorrer simultaneamente, mas isso pode causar inconsistências, como:

- **Atualização perdida:** Operações intercaladas corrompem dados.
- **Atualização temporária:** Falha de transação altera dados antes de restaurá-los.
- **Resumo incorreto:** Função de agregação afeta dados alterados por outras transações.
- **Leitura não repetitiva:** Leitura do mesmo item altera entre leituras consecutivas.

Esses problemas surgem quando as transações não são controladas adequadamente.

Processamento de Transações

Quando o SGBD processa uma transação, todas as operações devem ser completadas com sucesso para que as alterações sejam gravadas permanentemente. Se houver falha em alguma operação, a transação é cancelada e ocorre o *rollback*, restaurando os dados aos valores anteriores. Se a transação for bem-sucedida, usa-se o comando *commit* para confirmar as atualizações. As falhas possíveis durante uma transação incluem:

- Falha do computador
- Erro de transação ou sistema
- Condições de exceção detectadas
- Falha de disco ou catástrofes

Em seguida, serão apresentadas as propriedades das transações.

Propriedades de uma Transação

O SGBD garante a integridade dos dados com as propriedades ACID:

- **Atomicidade:** A transação deve ser totalmente realizada ou não executada.

- **Consistência:** A transação deve levar o banco de dados de um estado consistente a outro.
- **Isolamento:** A transação não deve ser interferida por outras.
- **Durabilidade:** As alterações no banco de dados devem persistir após a confirmação.

Estados de uma Transação

Uma transação pode passar pelos seguintes estados:

- **Ativo:** Início da transação, com operações de leitura e gravação.
- **Parcialmente confirmado:** Transação terminou, mas não confirmada.
- **Confirmado:** Mudanças gravadas permanentemente no banco de dados após verificação.
- **Falha:** Ocorre se houver erro ou abortamento durante o processamento.
- **Confirmado:** Transação sai do sistema após confirmação.

O SGBD mantém um log para registrar operações e permitir a recuperação de falhas.

Transações no PostgreSQL

Uma transação no PostgreSQL segue a estrutura:

- **BEGIN** – Inicia a transação.
- **COMMIT** – Confirma as alterações.
- **ROLLBACK** – Cancela a transação.
- **END** – Equivalente ao COMMIT.

CURSO		
CODIGOCURSO	int	PK
NOME	varchar(90)	
DATACRIACAO	date	N

Nos SGBDs, transações podem ser iniciadas implicitamente ao executar certos comandos. Por exemplo, um **INSERT** na tabela **CURSO** ocorre dentro de uma transação implícita:

```
-- BEGIN implícito
INSERT INTO CURSO (CODIGOCURSO, NOME, DATACRIACAO) VALUES (5, 'Engenharia de
Computação', NULL);
-- COMMIT implícito
```

Quando uma transação é desfeita, todas as operações envolvidas devem ser canceladas. No PostgreSQL, isso pode ser feito com **ROLLBACK**. Veja o exemplo de inserção na tabela **CURSO**, seguido do cancelamento da transação:

```
1 SELECT * FROM CURSO; -- Dados atuais
2 BEGIN;
3 INSERT INTO CURSO (CODIGOCURSO,NOME,DATACRIACAO) VALUES (6,'Psicologia',NULL);
4 SELECT * FROM CURSO; -- Dados após inserção
5 ROLLBACK;
6 END;
7 SELECT * FROM CURSO; -- Dados após desfazer a transação
```

Após o processamento do comando da linha 1, visualizaremos o conteúdo da tabela **CURSO** da seguinte maneira:

	123 codigocurso	ABC nome	T	datacriacao	T
1	2	Medicina		1990-05-10	
2	1	Sistemas de Informação		1999-06-19	
3	3	Nutrição		2012-02-19	
4	4	Pedagogia		1999-06-19	

Após o processamento dos comandos da linha 2 à linha 4, que já representam a inserção de um registro na tabela CURSO sob o contexto de uma transação explícita, teremos este resultado:

	123 codigocurso	ABC nome	T	datacriacao	T
1	2	Medicina		1990-05-10	
2	1	Sistemas de Informação		1999-06-19	
3	3	Nutrição		2012-02-19	
4	4	Pedagogia		1999-06-19	
5	6	Psicologia		[NULL]	

Finalmente, após o processamento dos comandos da linha 5 à linha 7, onde a transação é desfeita, teremos o resultado conforme a tabela a seguir:

	123 codigocurso	ABC nome	T	datacriacao	T
1	2	Medicina		1990-05-10	
2	1	Sistemas de Informação		1999-06-19	
3	3	Nutrição		2012-02-19	
4	4	Pedagogia		1999-06-19	

Perceba que o comando da linha 2 (BEGIN) iniciou explicitamente uma transação, a qual foi abortada quando da execução do comando da linha 5 (ROLLBACK).

No exemplo anterior, a transação envolveu apenas uma inserção na tabela **CURSO**, mas transações podem modificar várias linhas. A seguir, programamos uma transação com duas atualizações na tabela **DISCIPLINA**:

DISCIPLINA		
CODIGODISCIPLINA	int	PK
NOME	varchar(90)	
CARGAHORARIA	int	

Inicialmente, vamos listar o conteúdo da tabela DISCIPLINA. Podemos, então, usar o comando a seguir:

```
SELECT * FROM DISCIPLINA;
```

Após o processamento do comando, teremos o seguinte resultado:

	123 codigodisciplina	ABC nome	123 cargahoraria
1	1	Leitura e Produção de Textos	60
2	2	Redes de Computadores	60
3	3	Computação Gráfica	40
4	4	Anatomia	60

Agora, nossa intenção é alterar a carga horária das disciplinas de acordo com os critérios a seguir:

Disciplinas que possuem 60 horas: aumento em 20%

Disciplinas que possuam 40 horas: aumento em 10%

O código a seguir expressa uma transação envolvendo operações de atualização de dados na tabela DISCIPLINA:

```

1 BEGIN;
2 UPDATE DISCIPLINA SET CARGAHORARIA=CARGAHORARIA*1.2 WHERE CARGAHORARIA=60;
3 SELECT * FROM DISCIPLINA;
4 UPDATE DISCIPLINA SET CARGAHORARIA=CARGAHORARIA*1.1 WHERE CARGAHORARIA=40;
5 SELECT * FROM DISCIPLINA;
6 COMMIT;
```

Após a execução do trecho entre as linhas 1 e 3, o conteúdo da tabela disciplina estará conforme a seguir:

	123 codigodisciplina	ABC nome	123 cargahoraria
1	3	Computação Gráfica	40
2	1	Leitura e Produção de Textos	72
3	2	Redes de Computadores	72
4	4	Anatomia	72

Outro ponto interessante no projeto de transações é a utilização de pontos de salvamento (SAVEPOINT). Observe o exemplo a seguir:

```

1 BEGIN;
2 UPDATE DISCIPLINA SET CARGAHORARIA=CARGAHORARIA*1.2 WHERE CARGAHORARIA=60;
3 SELECT * FROM DISCIPLINA;
4 SAVEPOINT CARGA60;
5 UPDATE DISCIPLINA SET CARGAHORARIA=CARGAHORARIA*1.1 WHERE CARGAHORARIA=40;
6 ROLLBACK TO CARGA60;
7 SELECT * FROM DISCIPLINA;
8 COMMIT;
```

Na linha 4, adicionamos um SAVEPOINT denominado CARGA60. Quando a linha 6 for executada, o SGBD vai desfazer a operação de UPDATE da linha 5.

Uma transação não deve interferir em outra. No exemplo, após a execução do segundo **UPDATE**, apenas a disciplina **Computação Gráfica** permaneceu inalterada. Se outra aplicação consultasse a tabela **DISCIPLINA** nesse momento, veria os dados originais, pois as alterações ainda não foram confirmadas. Isso evita o problema de **atualização temporária**, garantindo que, se a transação for desfeita, todas as modificações sejam revertidas.

Uma transação que não modifica dados é chamada de **READ ONLY**, enquanto uma que pode modificar é **READ WRITE**. No PostgreSQL, o padrão ao iniciar uma transação é **READ WRITE**, mas é possível especificar o tipo com **SET TRANSACTION <TIPOTRANSAÇÃO>**.

Se uma transação **READ ONLY** tentar executar um comando de atualização, o SGBD retornará um erro, informando que essa ação não é permitida.

Ao longo do módulo, foi abordado o conceito de transação, que consiste em uma sequência de comandos executados integralmente ou desfeitos. Também foi explicado que o PostgreSQL cria implicitamente uma transação ao processar comandos de inserção, atualização ou remoção de dados. Por fim, foram apresentados comandos para gerenciar transações no PostgreSQL.

```
1 BEGIN;
2 SET TRANSACTION READ ONLY;
3 UPDATE DISCIPLINA SET CARGAHORARIA=80;
4 ROLLBACK;
```

Estrutura básica de uma comando SELECT

O comando **SELECT** é fundamental para manipular dados em bancos de dados, permitindo a seleção de informações específicas.

O vídeo explora o **modelo relacional**, base dos bancos de dados relacionais, abordando **tabelas, campos, chaves primárias e estrangeiras** e exemplos práticos de modelagem de dados.

O **SELECT** exibe os dados resultantes de uma consulta, podendo incluir colunas físicas, colunas calculadas ou expressões e funções. A apostila apresenta a sintaxe básica do comando e seus detalhes.

```
SQL (SQLite 3.27.2)
SELECT COLUNA1 [[AS] APELIDOCOLUNA1],
       COLUNA2 [[AS] APELIDOCOLUNA2],
       -
       COLUNAN [[AS] APELIDOCOLUNAN]
FROM TABELA;
```

A sintaxe apresentada é simplificada para facilitar o entendimento, enquanto a sintaxe completa do PostgreSQL oferece mais recursos para consultas complexas.

O comando **SELECT** pode ser utilizado de diferentes formas para obter o mesmo resultado e corresponde à operação de **projeção** na álgebra relacional.

Para exibir todas as colunas de uma tabela, usa-se **SELECT * FROM TABELA;**. No PostgreSQL, há uma forma simplificada: **TABLE tabela;**.

A seguir, serão explorados exemplos práticos baseados na tabela **ALUNO**.

ALUNO		
CODIGOALUNO	int	PK
NOME	varchar(90)	
SEXO	char(1)	
DTNASCIMENTO	date	
EMAIL	varchar(50)	N

Crie a tabela e insira dados usando o script a seguir na ferramenta de sua escolha. Certifique-se de estar conectado ao PostgreSQL e acessando um banco de dados criado por você.

Exibir todas as informações dos alunos.

`SELECT * FROM ALUNO;`

A tabela a seguir apresenta os resultados da consulta.

Resultados da consulta 01.

Ao executar a consulta, o SGBD percorre todos os registros da tabela ALUNO e exibe as colunas dessa tabela.

```
CREATE TABLE ALUNO (
    CODIGOALUNO int NOT NULL,
    NOME varchar(90) NOT NULL,
    SEXO char(1) NOT NULL,
    DTNASCIMENTO date NOT NULL,
    EMAIL varchar(30) NULL,
    CONSTRAINT ALUNO_pk PRIMARY KEY (CODIGOALUNO));
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO, EMAIL)
VALUES (1,'JOSÉ FRANCISCO TERRA', 'M', '28/10/1989', 'JFT@GMAIL.COM');
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO,EMAIL)
VALUES (2, 'ANDREY COSTA FILHO', 'M', '20/10/1999',
'ANDREYCF@HOTMAIL. COM');
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO, EMAIL)
VALUES (3, 'PATRÍCIA TORRES LOUREIRO', 'F', '20/10/1980',
'PTORRES@GMAIL.COM');
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO, EMAIL)
VALUES (4, 'CARLA MARIA MACIEL', 'F','20/11/1996', NULL);
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO,EMAIL)
VALUES (5, 'LEILA SANTANA COSTA', 'F','20/11/2001',NULL);
```

	123 códigoaluno	abc nome	abc sexo	⌚ dtnascimento	abc email	
1		JOSÉ FRANCISCO TERRA	M	1989-10-28	JFT@GMAIL.COM	
2		2 ANDREY COSTA FILHO	M	1999-10-20	ANDREYCF@HOTMAIL.COM	
3		3 PATRÍCIA TORRES LOUREIRO	F	1980-10-20	PTORRES@GMAIL.COM	
4		4 CARLA MARIA MACIEL	F	1996-11-20	[NULL]	
5		5 LEILA SANTANA COSTA	F	2001-11-20	[NULL]	

Retornar o código, o nome e a data de nascimento de todos os alunos.

`SELECT CODIGOALUNO, NOME, DTNASCIMENTO FROM ALUNO;`

Resultados da consulta 02.

Na consulta 02, foram especificadas três colunas da tabela ALUNO para serem exibidas ao usuário.

	123 códigoaluno	abc nome	⌚ dtnascimento
1	1	JOSÉ FRANCISCO TERRA	1989-10-28
2	2	ANDREY COSTA FILHO	1999-10-20
3	3	PATRÍCIA TORRES LOUREIRO	1980-10-20
4	4	CARLA MARIA MACIEL	1996-11-20
5	5	LEILA SANTANA COSTA	2001-11-20

Em especial, pode ser interessante renomear as colunas resultantes da consulta, visando tornar os resultados mais apresentáveis ao usuário da aplicação. Por exemplo, a consulta 02 pode ser reescrita da seguinte forma:

```
SQL (SQLite 3.27.2)
SELECT CODIGOALUNO AS "Matrícula", NOME AS "Nome do discente",
DTNASCIMENTO AS "Data de nascimento"
FROM ALUNO;
```

O resultado dessa consulta seria este:

	123 Matrícula	abc Nome do discente	⌚ Data de nascimento
1	1	JOSÉ FRANCISCO TERRA	1989-10-28
2	2	ANDREY COSTA FILHO	1999-10-20
3	3	PATRÍCIA TORRES LOUREIRO	1980-10-20
4	4	CARLA MARIA MACIEL	1996-11-20
5	5	LEILA SANTANA COSTA	2001-11-20

Os nomes das colunas exibidos na consulta podem não existir fisicamente no banco de dados. Nem toda coluna no resultado de uma consulta corresponde a uma coluna real de uma tabela.

As **funções de data e hora** permitem manipular e analisar dados temporais em bancos de dados. Com elas, é possível realizar cálculos entre datas, extrair partes específicas e fazer manipulações avançadas, essenciais para análises temporais.

No PostgreSQL, algumas funções importantes incluem:

- **current_date**: retorna a data atual.
- **current_time**: retorna a hora atual.
- **current_timestamp**: retorna data e hora atuais.
- **extract(campo from fonte)**: extrai subcampos como século, ano, mês e dia.

Mais detalhes podem ser encontrados na documentação oficial do PostgreSQL.

```
SQL (SQLite 3.27.2)
SELECT CURRENT_DATE AS "Data Atual",
       CURRENT_TIME AS "Hora Atual",
       CURRENT_TIMESTAMP "Data e Hora atuais",
       EXTRACT( DOY FROM CURRENT_DATE) AS "Dia do ano",
       -- DOW 0 - domingo, 1 - segunda, ..., 6 - sábado
       EXTRACT( DOW FROM CURRENT_DATE) AS "Dia da semana",
       EXTRACT( DAY FROM CURRENT_DATE) AS "Dia Atual",
       EXTRACT( MONTH FROM CURRENT_DATE) AS "Mês Atual",
       EXTRACT( YEAR FROM CURRENT_DATE) AS "Ano Atual",
       EXTRACT( CENTURY FROM CURRENT_DATE) AS "Século Atual";
```

Agora veja na tabela os resultados da consulta!

	Data Atual	Hora Atual	Data e Hora atuais	Dia do ano	Dia da semana	Dia Atual	Mês Atual	Ano Atual	Século Atual
1	2020-06-24	08:55:13	2020-06-24 08:55:13	176	3	24	6	2.020	21

Utilizamos o qualificador **AS "Apelido"** para facilitar a interpretação dos resultados das funções. Não há cláusula **FROM**, pois as colunas retornadas vêm de funções do PostgreSQL, não de uma tabela.

No padrão SQL, a cláusula **FROM** é obrigatória, mas no PostgreSQL, é possível executar um **SELECT** sem ela (exemplo: **SELECT 5+5**).

Para exibir o nome do dia da semana, veja o código a seguir.

```
SQL (SQLite 3.27.2)
SELECT CASE WHEN extract(dow from current_date) = 0 THEN 'domingo'
            WHEN extract(dow from current_date) = 1 THEN 'segunda-feira'
            WHEN extract(dow from current_date) = 2 THEN 'terça-feira'
            WHEN extract(dow from current_date) = 3 THEN 'quarta-feira'
            WHEN extract(dow from current_date) = 4 THEN 'quinta-feira'
            WHEN extract(dow from current_date)=5 THEN 'sexta-feira'
            WHEN extract(dow from current_date)=6 THEN 'sábado'
        END AS "Nome do dia da semana";
```

Utilizamos o comando **CASE**, equivalente ao **IF**, com a cláusula **WHEN** para avaliar expressões que retornam um inteiro representativo do dia da semana, quando a expressão for verdadeira.

Calculando idade e faixa etária

Para calcular **idade** e **faixa etária**, é comum extrair essas informações de uma coluna com data de nascimento. O código a seguir retorna nome, data de nascimento e idade dos alunos.

```
SQL (SQLite 3.27.2)
SELECT NOME,
       DTNASCIMENTO,
       AGE(DTNASCIMENTO) AS "Idade [ano/mês/dia]",
       EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) AS "Idade do Aluno"
  FROM ALUNO;
```

	ABC nome	dtnascimento	Idade [ano/mês/dia]	123 Idade do Aluno
1	JOSÉ FRANCISCO TERRA	1989-10-28	30 years 7 mons 30 days	30
2	ANDREY COSTA FILHO	1999-10-20	20 years 8 mons 7 days	20
3	PATRÍCIA TORRES LOUREIRO	1980-10-20	39 years 8 mons 7 days	39
4	CARLA MARIA MACIEL	1996-11-20	23 years 7 mons 7 days	23
5	LEILA SANTANA COSTA	2001-11-20	18 years 7 mons 7 days	18

Muito bem, agora, vamos exibir o nome, a idade e a faixa etária dos alunos.

Observe o código SQL a seguir.

```
SQL (SQLite 3.27.2)

SELECT nome,
       EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) AS "Idade do Aluno",
       CASE WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) <=20 THEN '1. até 20 anos'
             WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 21 AND 30 THEN '2. 21 a 30 anos'
             WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 31 AND 40 THEN '3. 31 a 40 anos'
             WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 41 AND 50 THEN '4. 41 a 50 anos'
             WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 51 AND 60 THEN '5. 51 a 60 anos'
             WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) > 60 THEN '6. mais de 60 anos'
       END AS "Faixa Etária"
FROM ALUNO;
```

Perceba que cada linha com a cláusula WHEN avalia a expressão que retorna uma faixa etária de acordo com a idade do aluno.

A seguir, o resultado da consulta. Confira!

	ABC nome	123 Idade do Aluno	ABC Faixa Etária
1	JOSÉ FRANCISCO TERRA	30	2. 21 a 30 anos
2	ANDREY COSTA FILHO	20	1. até 20 anos
3	PATRÍCIA TORRES LOUREIRO	39	3. 31 a 40 anos
4	CARLA MARIA MACIEL	23	2. 21 a 30 anos
5	LEILA SANTANA COSTA	18	1. até 20 anos

As **funções de resumo ou agregação** permitem analisar grandes conjuntos de dados, com funções como **SUM**, **AVG**, **COUNT**, entre outras, para calcular estatísticas e extrair insights. Esse conhecimento é essencial para criar relatórios e análises em sistemas de banco de dados.

Funções úteis para resumo de dados:

- **COUNT(*)**: número de linhas.
- **MIN(COLUNA/EXPRESSÃO)**: menor valor.
- **AVG(COLUNA/EXPRESSÃO)**: média dos valores.
- **MAX(COLUNA/EXPRESSÃO)**: maior valor.
- **SUM(COLUNA/EXPRESSÃO)**: soma dos valores.
- **STDDEV(COLUNA/EXPRESSÃO)**: desvio-padrão.
- **VARIANCE(COLUNA/EXPRESSÃO)**: variância.

SQL (SQLite 3.27.2)

```
SELECT
    COUNT(*) AS "Número de alunos",
    MIN(EXTRACT(YEAR FROM AGE(DTNASCIMENTO))) AS "Menor Idade",
    AVG(EXTRACT(YEAR FROM AGE(DTNASCIMENTO))) AS "Idade Média",
    MAX(EXTRACT(YEAR FROM AGE(DTNASCIMENTO))) AS "Maior Idade",
    SUM(EXTRACT(YEAR FROM AGE(DTNASCIMENTO)))/COUNT(*) AS "Idade Média"
FROM ALUNO;
```

Perceba que, como estamos usando somente o comando SELECT/FROM, cada função é calculada levando em consideração todos os registros da tabela.

Veja na imagem a seguir o resultado da consulta.

	1 Número de alunos	1 Menor Idade	1 Idade Média	1 Maior Idade	1 Idade Média
1	5	18	26	39	26

Listando resumos em uma linha

Suponha que haja interesse em conhecer os quantitativos de cursos, disciplinas e alunos do nosso banco de dados. Poderíamos submeter ao SGBD as consultas a seguir.

Curso

```
SELECT COUNT(*) NCURSOS FROM CURSO;
```

Disciplina

```
SELECT COUNT(*) NDISCIPINAS FROM DISCIPLINA;
```

Aluno

```
SELECT COUNT(*) NALUNOS FROM ALUNO;
```

Estamos diante de três consultas. No entanto, pode ser mais interessante mostrarmos os resultados em apenas uma linha. Podemos, então, submeter o seguinte código:

SQL (SQLite 3.27.2)

```
SELECT
    (SELECT COUNT(*) NCURSOS FROM CURSO),
    (SELECT COUNT(*) NALUNOS FROM ALUNO),
    (SELECT COUNT(*) NDISCIPINAS FROM DISCIPLINA);
```

O que fizemos?

Como cada consulta (linhas 2 a 4) retorna somente um valor, utilizamos um SELECT externo (linha 1) para exibir cada coluna resultante.

Observe o resultado!

	1 ncursos	1 nalunos	1 ndiscipinas
1	4	5	4

Convém ressaltar que o comando é válido, visto que, no PostgreSQL, a cláusula FROM não é obrigatória.

A criação de tabelas e views a partir de consultas SQL é um processo fundamental para organizar dados e facilitar o acesso e análise em projetos de banco de dados. Abaixo, detalho os conceitos e exemplos de como criar tabelas e views a partir de uma consulta SQL.

1. Criando Tabela a Partir de Consulta

Quando você deseja armazenar os resultados de uma consulta SQL em uma nova tabela, o comando utilizado é o `CREATE TABLE`. Ele permite criar uma tabela e inserir nela os dados resultantes de uma consulta.

```
SQL (SQLite 3.27.2)

CREATE TABLE TTESTE AS
SELECT NOME,
       EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) AS "Idade do Aluno",
       CASE WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) <=20 THEN '1. até 20 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 21 AND 30 THEN '2. 21 a 30 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 31 AND 40 THEN '3. 31 a 40 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 41 AND 50 THEN '4. 41 a 50 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 51 AND 60 THEN '5. 51 a 60 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) > 60 THEN '6. mais de 60 anos'
       END AS "Faixa Etária"
FROM ALUNO;
```

Neste exemplo, a tabela `TTESTE` será criada e armazenará os dados de todos os funcionários com salário superior a 5000.

2. Criando View a Partir de Consulta

Uma **view** é uma consulta armazenada no banco de dados que pode ser tratada como uma tabela. Ela não armazena dados de forma permanente, mas sim a consulta que gera esses dados. Views são úteis para simplificar consultas complexas e fornecer uma interface de acesso aos dados de forma personalizada.

```
SQL (SQLite 3.27.2)

CREATE VIEW VTESTE AS
SELECT NOME,
       EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) AS "Idade do Aluno",
       CASE WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) <=20 THEN '1. até 20 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 21 AND 30 THEN '2. 21 a 30 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 31 AND 40 THEN '3. 31 a 40 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 41 AND 50 THEN '4. 41 a 50 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) BETWEEN 51 AND 60 THEN '5. 51 a 60 anos'
            WHEN EXTRACT(YEAR FROM AGE(DTNASCIMENTO)) > 60 THEN '6. mais de 60 anos'
       END AS "Faixa Etária"
FROM ALUNO;
```

Após a criação da view `VTESTE`, você pode consultar os dados armazenados nela da seguinte maneira:

```
SELECT * FROM VTESTE;
```

Isso resultará na execução da consulta original definida para a view.

Diferenças entre Tabela e View

- **Tabela:** Armazena fisicamente os dados no banco de dados.
- **View:** Não armazena dados fisicamente, mas executa uma consulta que retorna dados sob demanda.

Benefícios

- **Criar tabelas** a partir de uma consulta permite persistir um conjunto de dados para análise futura.
- **Criar views** facilita a organização de consultas complexas e oferece uma maneira de encapsular a lógica de consulta, simplificando o acesso aos dados.

Essas técnicas são úteis para otimizar o desempenho e organizar dados de forma estruturada em bancos de dados, melhorando a eficiência na análise e na manipulação das informações.

A cláusula WHERE permite filtrar registros com base em condições específicas.

Neste vídeo, aprenderemos a usar o WHERE para realizar consultas SQL, filtrando dados de forma precisa e eficiente.

ALUNO		
CODIGOALUNO	int	PK
NOME	varchar(90)	
SEXO	char(1)	
DTNASCIMENTO	date	
EMAIL	varchar(50)	N

Recomendamos criar uma tabela e inserir algumas linhas usando o script abaixo, a partir da ferramenta de sua escolha. Certifique-se de estar conectado ao PostgreSQL e acessando um banco de dados criado por você.

```
CREATE TABLE ALUNO (
    CODIGOALUNO int NOT NULL,
    NOME varchar(90) NOT NULL,
    SEXO char(1) NOT NULL,
    DTNASCIMENTO date NOT NULL,
    EMAIL varchar(30) NULL,
    CONSTRAINT ALUNO_pk PRIMARY KEY (CODIGOALUNO));
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO, EMAIL)
VALUES (1, 'JOSÉ FRANCISCO TERRA', 'M', '28/10/1989', 'JFT@GMAIL.COM');
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO,EMAIL)
VALUES (2, 'ANDREY COSTA FILHO', 'M','20/10/1999','ANDREYCF@HOTMAIL.COM');
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO,EMAIL)
VALUES (3,'PATRICIA TORRES LOUREIRO','F','20/10/1980', 'PTORRES@GMAIL.COM');
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO,EMAIL)
VALUES (4, 'CARLA MARIA MACIEL', 'F','20/11/1996',NULL);
INSERT INTO ALUNO (CODIGOALUNO, NOME, SEXO, DTNASCIMENTO,EMAIL)
VALUES (5, 'LEILA SANTANA COSTA', 'F','20/11/2001',NULL);
```

Recuperando dados com select/from/where/order by

Uma sintaxe básica para o comando SELECT, com o uso das cláusulas WHERE e ORDER BY, está expressa a seguir. Veja!

```
SQL (SQLite 3.27.2)
SELECT COLUNA1 [[AS] APELIDOCOLUNA1],
COLUNA2 [[AS] APELIDOCOLUNA2],
...
COLUNAN [[AS] APELIDOCOLUNAN]
FROM TABELA
WHERE <CONDIÇÃO>
ORDER BY EXPRESSÃO1[ASC|DESC] [NULLS \{FIRST|LAST\}], [EXPRESSÃO2[ASC|DESC] [NULLS \
{FIRST|LAST\}...];]
```

O comando **SELECT** define as colunas a serem retornadas na consulta, enquanto o **FROM** especifica a tabela a ser consultada. A cláusula **WHERE** aplica condições (simples ou compostas) para filtrar os registros retornados pelo SGBD, realizando a operação de seleção da álgebra relacional. No **ORDER BY**, é possível ordenar os resultados com base em uma ou mais colunas, definindo a posição dos valores **NULL**.

As condições na cláusula **WHERE** utilizam operadores relacionais como:

- < (menor),
- <= (menor ou igual a),
- > (maior),
- >= (maior ou igual a),
- = (igual),
- <> ou != (diferente).

Além disso, operadores lógicos como **AND** (conjunção), **OR** (disjunção) e **NOT** (negação) podem ser usados para combinar ou negar condições.

Consulta 01 + resultado

Para mostrar o nome e a data de nascimento das professoras, observe o exemplo da consulta no código:

```
SQL (SQLite 3.27.2)
SELECT NOME, DTNASCIMENTO
FROM ALUNO
WHERE SEXO='F';
```

	ABC nome	dtnascimento
1	PATRÍCIA TORRES LOUREIRO	1980-10-20
2	CARLA MARIA MACIEL	1996-11-20
3	LEILA SANTANA COSTA	2001-11-20

Perceba que foi criada uma condição simples de igualdade envolvendo a coluna **SEXO** da tabela **ALUNO**. O SGBD percorre cada registro da tabela **ALUNO**, avalia a condição (linha 3) e exibe as colunas **NOME** e **DTNASCIMENTO** para cada registro cuja avaliação da condição retorne verdadeiro.

Consulta 02 + resultado

Para mostrar o nome e a data de nascimento das professoras que fazem aniversário em novembro, observe o exemplo:

```
SQL (SQLite 3.27.2)
SELECT NOME, DTNASCIMENTO
FROM ALUNO
WHERE SEXO='F' AND EXTRACT (MONTH FROM DTNASCIMENTO)=11;
```

	ABC nome	dtnascimento
1	CARLA MARIA MACIEL	1996-11-20
2	LEILA SANTANA COSTA	2001-11-20

Perceba que foi criada uma condição composta envolvendo uma conjunção. O SGBD retornará os registros que possuem o valor “F” para a coluna SEXO e o inteiro 11 como valor do mês referente à data de nascimento.

Recuperando dados com o uso dos operadores IN e BETWEEN

O operador **IN** permite filtrar resultados com base em uma lista de valores, enquanto o **BETWEEN** seleciona registros dentro de um intervalo específico. O operador **[NOT] IN** é usado para comparar dados com uma lista de valores.

Consulta 03 + resultado

Para listar o nome dos alunos que fazem aniversário no segundo semestre, observe o código:

	ABC nome
1	JOSÉ FRANCISCO TERRA
2	ANDREY COSTA FILHO
3	CARLA MARIA MACIEL
4	LEILA SANTANA COSTA

Note que a expressão na cláusula WHERE compara o mês de nascimento de cada aluno com os valores da lista contendo os inteiros correspondentes aos meses do segundo semestre.

Recuperando dados com o uso do operador BETWEEN

O operador **[NOT] BETWEEN** verifica se determinado valor encontra-se no intervalo entre dois valores.

Exemplo

$X \text{ BETWEEN } Y \text{ AND } Z$ é equivalente a $X \geq Y \text{ AND } X \leq Z$. De modo semelhante, $X \text{ NOT BETWEEN } Y \text{ AND } Z$ é equivalente a $X < Y \text{ OR } X > Z$.

Consulta 04 + resultado

Para listar o nome dos alunos nascidos entre 1985 e 2005, observe o exemplo:

	ABC nome
1	ANDREY COSTA FILHO
2	LEILA SANTANA COSTA

Note que a expressão na cláusula WHERE compara o ano de nascimento de cada aluno junto com o intervalo especificado pelo operador BETWEEN. Caso quiséssemos extrair o mesmo resultado sem o uso do BETWEEN, poderíamos programar um comando equivalente, conforme a seguir.

```
SQL (SQLite 3.27.2)
SELECT NOME
FROM ALUNO
WHERE EXTRACT (YEAR FROM DTNASCIMENTO) >= 1985 AND EXTRACT (YEAR FROM DTNASCIMENTO) <= 2005;
```

O operador **LIKE** permite buscar registros que correspondem a padrões específicos de texto, utilizando curingas para tornar a pesquisa mais flexível. É amplamente utilizado em buscas textuais em bancos de dados.

O **LIKE** funciona com dois símbolos especiais:

- O **_** substitui um único caractere.
- O **%** substitui qualquer sequência de caracteres.

Além disso, o operador **[NOT] LIKE** pode ser usado para excluir padrões específicos.

Exemplo:

- **LIKE 'a%**' encontra registros que começam com "a".
- **LIKE '_a%**' encontra registros em que o segundo caractere seja "a".
- **LIKE '%abc'** encontra registros que terminam com "abc".

Essas ferramentas são úteis para realizar buscas detalhadas e flexíveis em dados textuais.

Consulta 05 + resultado

Para listar o nome dos alunos que possuem a string COSTA em qualquer parte do nome, veja o código:

```
SQL (SQLite 3.27.2)
SELECT NOME
FROM ALUNO
WHERE NOME LIKE "%COSTA%";
```

ABC nome	
1	ANDREY COSTA FILHO
2	LEILA SANTANA COSTA

O uso do padrão '**%COSTA%**' significa que não importa o conteúdo localizado antes e depois da string "COSTA".

Consulta 06 + resultado

Para listar o nome dos alunos que possuem a letra "A" na segunda posição do nome, veja o exemplo de consulta:

```
SQL (SQLite 3.27.2)
SELECT NOME
FROM ALUNO
WHERE NOME LIKE '_A%';
```

ABC nome	
1	PATRÍCIA TORRES LOUREIRO
2	CARLA MARIA MACIEL

Note que, para especificar o “A” na segunda posição, o SGBD desprezará qualquer valor na primeira posição da string, não importando o que estiver localizado à direita do “A”.

Consulta 07 + resultado

Para listar o nome e a data de nascimento dos alunos que não possuem a string “MARIA” fazendo parte do nome, veja o exemplo de consulta a seguir:

SQL (SQLite 3.27.2)		ABC nome	dtnascimento
1	SELECT NOME, DTNASCIMENTO FROM ALUNO WHERE NOME NOT LIKE '%MARIA%';	JOSÉ FRANCISCO TERRA	1989-10-28
2		ANDREY COSTA FILHO	1999-10-20
3		PATRÍCIA TORRES LOUREIRO	1980-10-20
4		LEILA SANTANA COSTA	2001-11-20

Estamos diante de um caso semelhante ao da consulta 05.

No entanto, utilizamos o operador de negação para retornar os registros de interesse.

Consulta 08 + resultado

Quantos alunos possuem conta de e-mail no Gmail? Veja o exemplo de consulta na imagem a seguir.

SQL (SQLite 3.27.2)		123 quantidade
1	SELECT COUNT(*) AS QUANTIDADE FROM ALUNO WHERE EMAIL LIKE '%@GMAIL.%';	2

Note que, mais uma vez, estamos diante de um caso semelhante ao da consulta 05. Buscamos pela string “@GMAIL.” em qualquer posição da coluna EMAIL.

O operador **IS NULL** é usado para identificar registros sem valores em colunas opcionais, indicando que o dado é “desconhecido” ou “não aplicável”. Para verificar se uma coluna possui valor, utiliza-se **IS NOT NULL**.

A cláusula **ORDER BY** permite classificar os resultados da consulta, organizando os dados conforme um critério específico.

Consulta 09 + resultado

Para listar o nome, a data de nascimento e o e-mail dos alunos que têm endereço eletrônico cadastrado, veja o exemplo de consulta a seguir.

SQL (SQLite 3.27.2)		ABC nome	dtnascimento	ABC email
1	SELECT NOME, DTNASCIMENTO, EMAIL FROM ALUNO 3 WHERE EMAIL IS NOT NULL;	JOSÉ FRANCISCO TERRA	1989-10-28	JFT@GMAIL.COM
2		ANDREY COSTA FILHO	1999-10-20	ANDREYCF@HOTMAIL.COM
3		PATRÍCIA TORRES LOUREIRO	1980-10-20	PTORRES@GMAIL.COM

O SGBD retorna os registros onde há algum conteúdo cadastrado na coluna EMAIL.

Consulta 10 + resultado

Para retornar o nome dos alunos sem e-mail cadastrado no banco de dados, veja o exemplo:

SQL (SQLite 3.27.2)	
<pre>SELECT NOME FROM ALUNO WHERE EMAIL IS NULL;</pre>	
1	CARLA MARIA MACIEL
2	LEILA SANTANA COSTA

O SGBD retorna os registros sobre os quais não há valor cadastrado na coluna EMAIL.

Recuperando dados usando ordenação dos resultados

Para melhor organizar os resultados de uma consulta, podemos especificar critérios de ordenação. Vejamos alguns exemplos

Consulta 11 + resultado

Para retornar o nome e a data de nascimento dos alunos, ordenando os resultados por nome, de maneira ascendente, veja o exemplo:

SQL (SQLite 3.27.2)	
<pre>1 SELECT NOME, DTNASCIMENTO 2 FROM ALUNO 3 ORDER BY NOME;</pre>	
1	ANDREY COSTA FILHO
2	CARLA MARIA MACIEL
3	JOSÉ FRANCISCO TERRA
4	LEILA SANTANA COSTA
5	PATRÍCIA TORRES LOUREIRO

O SGBD retorna os registros da tabela ALUNO, obedecendo ao critério de ordenação especificado na linha 3 da consulta. O padrão ascendente (ASC) é opcional.

Consulta 12 + resultado

Para retornar o nome e a data de nascimento dos alunos, ordenando os resultados de modo ascendente pelo mês de nascimento e, em seguida, pelo nome, também de modo ascendente.,veja o exemplo:

SQL (SQLite 3.27.2)	
<pre>1 SELECT NOME, DTNASCIMENTO 2 FROM ALUNO 3 ORDER BY EXTRACT(MONTH FROM DTNASCIMENTO), NOME;</pre>	
1	ANDREY COSTA FILHO
2	JOSÉ FRANCISCO TERRA
3	PATRÍCIA TORRES LOUREIRO
4	CARLA MARIA MACIEL
5	LEILA SANTANA COSTA

O SGBD retorna os registros da tabela ALUNO, levando em conta o critério de ordenação especificado na linha 3 da consulta. Foi realizada ordenação pelo mês de nascimento; em seguida, pelo nome.

Aprenderemos a usar a cláusula GROUP BY para agrupar registros por valores comuns e aplicar funções de agregação para calcular estatísticas desses grupos.

Consultas com GROUP BY e HAVING

Em nossas consultas, usaremos como base a tabela FUNCIONARIO, conforme imagem a seguir.

FUNCIONARIO		
CODIGOFUNCIONARIO	int	PK
NOME	varchar(90)	
CPF	char(15)	
SEXO	char(1)	
DTNASCIMENTO	date	
SALARIO	real	N

Crie a tabela e insira linhas usando o script abaixo na ferramenta de sua escolha. Certifique-se de estar conectado ao PostgreSQL e acessando um database criado por você.

```
CREATE TABLE FUNCIONARIO (
    CODIGOFUNCIONARIO int NOT NULL,
    NOME varchar(90) NOT NULL,
    CPF char(15) NULL,
    SEXO char(1) NOT NULL,
    DTNASCIMENTO date NOT NULL,
    SALARIO real NULL,
    CONSTRAINT FUNCIONARIO_pk PRIMARY KEY (CODIGOFUNCIONARIO));
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO) VALUES (1, 'ROBERTA SILVA BRASIL', '82998', 'F', '20/02/1980', 7000);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO) VALUES (2, 'MARIA SILVA BRASIL', '9876', 'F', '20/09/1988', 9500);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO) VALUES (3, 'GABRIELLA PEREIRA LIMA', '32998', 'F', '20/02/1990', 6000);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO) VALUES (4, 'MARCOS PEREIRA BRASIL', '9999', 'M', '20/02/1999', 6000);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO) VALUES (5, 'HEMERSON SILVA BRASIL', '9111', 'M', '20/12/1992', 4000);
```

Após a criação da tabela e a inserção dos registros, podemos utilizar o código a seguir para exibir todo o seu conteúdo.

```
SQL (SQLite 3.27.2)
1 SELECT *
2 FROM FUNCIONARIO;
```

O resultado da consulta será semelhante à tabela a seguir.

	codigofuncionario	nome	cpf	sexo	dtnascimento	salario
1	1	ROBERTA SILVA BRASIL	[NULL]	F	1980-02-20	7000.0
2	2	MARIA SILVA BRASIL	[NULL]	F	1988-09-20	9500.0
3	3	GABRIELLA PEREIRA LIMA	[NULL]	F	1990-02-20	6000.0
4	4	MARCOS PEREIRA BRASIL	[NULL]	M	1999-02-20	6000.0
5	5	HEMERSON SILVA BRASIL	[NULL]	M	1992-12-20	4000.0

Aprenderemos a projetar consultas usando agrupamento de dados com GROUP BY e HAVING,

essenciais para análises gerenciais. Essas consultas geralmente utilizam funções de resumo como SUM (soma), AVG (média), MIN (mínimo) e MAX (máximo).

Os valores de uma coluna podem formar grupos sobre os quais podemos recuperar dados. No exemplo, registros são agrupados pelo valor da coluna sexo, organizando os funcionários em categorias distintas. Para representar esses grupos, usamos chaves, embora essa estrutura não exista no modelo relacional.

Para exibir grupos no SQL, podemos utilizar a cláusula DISTINCT no comando SELECT, garantindo que cada grupo seja exibido corretamente.

SQL (SQLite 3.27.2)	
1	SELECT DISTINCT SEXO
2	FROM FUNCIONARIO;

	ABC sexo ↑↓
1	M
2	F

Vamos perceber que em SQL a cláusula mais adequada para trabalhar com agrupamento de dados é o GROUP BY.

A cláusula GROUP BY organiza os resultados da consulta conforme um grupo especificado. Ela é declarada após FROM ou, se houver, após WHERE. Por exemplo, para reproduzir o resultado do comando anterior, podemos usar o seguinte código:

SQL (SQLite 3.27.2)
1 SELECT SEXO
2 FROM FUNCIONARIO
3 GROUP BY SEXO;

No entanto, vamos perceber que o uso mais conhecido da cláusula GROUP BY ocorre quando associada a funções de agregação, tais como COUNT, MIN, MAX e AVG. Vamos estudar alguns exemplos!

Consulta 01 + resultado

Para retornar o número de funcionários por sexo, temos o seguinte código:

SQL (SQLite 3.27.2)	
1	SELECT SEXO, COUNT(*) AS QUANTIDADE
2	FROM FUNCIONARIO
3	GROUP BY SEXO;

	ABC sexo ↑↓	123 quantidade ↑↓
1	M	2
2	F	3

O SGBD realiza o agrupamento de dados de acordo com os valores da coluna SEXO. Em seguida, para cada grupo encontrado, a função COUNT(*) é executada e o resultado exibido.

E se tivéssemos interesse em exibir os resultados da consulta anterior em uma única linha? Poderíamos usar o código a seguir.

SQL (SQLite 3.27.2)
SELECT
(SELECT COUNT(*) AS "M" FROM FUNCIONARIO WHERE SEXO='M'),
(SELECT COUNT (*) AS "F" FROM FUNCIONARIO WHERE SEXO='F');

Consulta 02 + resultado

Para retornar a média salarial por sexo, temos o seguinte código:

```
SQL (SQLite 3.27.2)
SELECT SEXO,
       AVG(SALARIO) AS MEDIASALARIAL
  FROM FUNCIONARIO
 GROUP BY SEXO;
```

	sex	mediasalarial
1	M	5.000
2	F	7.500

O SGBD realiza o agrupamento de dados de acordo com os valores da coluna SEXO. Em seguida, para cada grupo encontrado, a função AVG (SALARIO) é executada; e o resultado, exibido.

Consulta 03 + resultado

Para retornar, por mês de aniversário, a quantidade de colaboradores, o menor salário, o maior salário e o salário médio, ordene os resultados por mês de aniversário, conforme o código.

```
SQL (SQLite 3.27.2)
```

```
SELECT EXTRACT(MONTH FROM DTNASCIMENTO) AS MES,
       COUNT(*) AS QUANTIDADE,
       MIN(SALARIO) AS MENORSALARIO,
       ROUND(AVG(SALARIO)::NUMERIC,0) AS SALARIOMEDIO,
       MAX(SALARIO) AS MAIORSALARIO
  FROM FUNCIONARIO
 GROUP BY EXTRACT(MONTH FROM DTNASCIMENTO)
 ORDER BY EXTRACT(MONTH FROM DTNASCIMENTO);
```

	mes	quantidade	menorsalario	salariomedio	maiorsalario
1	2	3	6.000	6.333	7.000
2	9	1	9.500	9.500	9.500
3	12	1	4.000	4.000	4.000

O SGBD agrupa os dados pelo mês de nascimento dos funcionários, aplica funções de agregação a cada grupo e exibe os resultados. A função ROUND na linha 4 é usada para exibir apenas a parte inteira da média salarial.

Consulta 04 + resultado

Para retornar, por mês de aniversário, o mês, o sexo e a quantidade de colaboradores, veja o código a seguir. Os resultados ordenados são apresentados pelo mês.

```
SQL (SQLite 3.27.2)
SELECT EXTRACT(MONTH FROM DTNASCIMENTO) AS MES,
       SEXO,
       COUNT(*) AS QUANTIDADE
  FROM FUNCIONARIO
 GROUP BY EXTRACT(MONTH FROM DTNASCIMENTO), SEXO
 ORDER BY EXTRACT(MONTH FROM DTNASCIMENTO);
```

	mes	sex	quantidade
1	2	F	2
2	2	M	1
3	9	F	1
4	12	M	1

O SGBD agrupa os dados pelo mês de aniversário e, dentro de cada mês, cria grupos por sexo. Para cada combinação mês/sexo, calcula o número de colaboradores.

Exploraremos técnicas avançadas de filtragem e agrupamento com GROUP BY e HAVING. A

cláusula HAVING permite aplicar condições de filtro a grupos de dados, enquanto o GROUP BY organiza os dados. Essa combinação é útil para análises detalhadas, especialmente em relatórios complexos.

Diferente da cláusula WHERE, que filtra dados simples, HAVING é usada quando o filtro envolve cálculos de funções de agregação. Vejamos um exemplo de seu uso.

Consulta 05 + resultado

Suponha que o departamento de recursos humanos esteja estudando a viabilidade de oferecer bônus de 5% aos funcionários por mês de nascimento, mas limitado somente aos casos em que há mais de um colaborador aniversariando. Assim, para cada mês em questão, deseja-se listar o mês, o número de colaboradores e o valor do bônus.

```
SQL (SQLite 3.27.2)
SELECT EXTRACT(MONTH FROM DTNASCIMENTO) AS MES,
       COUNT(*) AS QUANTIDADE,
       SUM(SALARIO*0.05) AS TOTALBONUS
  FROM FUNCIONARIO
 GROUP BY EXTRACT(MONTH FROM DTNASCIMENTO)
 HAVING COUNT(*)>1
 ORDER BY EXTRACT(MONTH FROM DTNASCIMENTO);
```

	mes	quantidade	totalbonus
1	2	3	950

A consulta é similar à consulta 03, mas agora queremos filtrar registros onde a coluna quantidade seja maior que 1. Como quantidade é uma coluna calculada com função de agregação, não podemos usar WHERE. Em vez disso, aplicamos o filtro com a cláusula HAVING, conforme mostrado na linha 6.

O uso de agrupamento de dados é essencial para extração de informações analíticas com funções de agregação. Quando o filtro depende de uma função de agregação, usamos a cláusula HAVING.

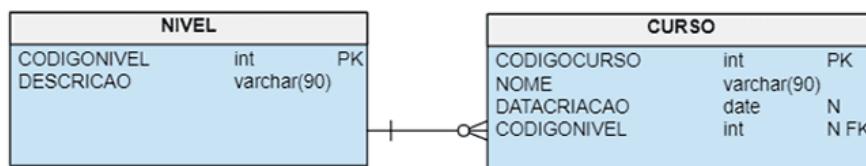
Junção e produto cartesiano

Quando precisamos retornar dados de diferentes tabelas, como o nome do departamento da tabela Departamento e o nome da região da tabela Região, realizamos uma junção.

A junção de tabelas é uma operação essencial no modelo relacional de banco de dados. Matematicamente, ela é um subconjunto do produto cartesiano entre as tabelas, especificado por uma condição de junção entre colunas.

Existem dois tipos principais de junção:

- **INNER JOIN** (junção interna)
- **OUTER JOIN** (junção externa), que pode ser LEFT, FULL ou RIGHT OUTER JOIN.



```

CREATE TABLE NIVEL (
    CODIGONIVEL int NOT NULL,
    DESCRICAO varchar(90) NOT NULL,
    CONSTRAINT CHAVEPNIVEL PRIMARY KEY (CODIGONIVEL));
CREATE TABLE CURSO (
    CODIGOCURSO int NOT NULL,
    NOME varchar(90) NOT NULL UNIQUE,
    DATAACRIACAO date NULL,
    CODIGONIVEL int NULL,
    CONSTRAINT CHAVEPCURSO PRIMARY KEY (CODIGOCURSO));
ALTER TABLE CURSO ADD FOREIGN KEY (CODIGONIVEL) REFERENCES NIVEL;
INSERT INTO NIVEL (CODIGONIVEL, DESCRICAO) VALUES (1,'Graduação');
INSERT INTO NIVEL (CODIGONIVEL, DESCRICAO) VALUES (2,'Especialização');
INSERT INTO NIVEL (CODIGONIVEL, DESCRICAO) VALUES (3,'Mestrado');
INSERT INTO NIVEL (CODIGONIVEL, DESCRICAO) VALUES (4,'Doutorado');
INSERT INTO CURSO (CODIGOCURSO,NOME,DATAACRIACAO, CODIGONIVEL) VALUES (1, 'Sistemas de Informação', '19/06/1999',1);
INSERT INTO CURSO (CODIGOCURSO,NOME,DATAACRIACAO, CODIGONIVEL) VALUES (2, 'Medicina', '10/05/1990',1);
INSERT INTO CURSO (CODIGOCURSO,NOME,DATAACRIACAO, CODIGONIVEL) VALUES (3, 'Nutrição', '19/02/2012',NULL);
INSERT INTO CURSO (CODIGOCURSO,NOME,DATAACRIACAO, CODIGONIVEL) VALUES (4, 'Pedagogia','19/06/1999',1);
INSERT INTO CURSO (CODIGOCURSO,NOME,DATAACRIACAO, CODIGONIVEL) VALUES (5, 'Saúde da Família','10/09/1999',3);
INSERT INTO CURSO (CODIGOCURSO,NOME,DATAACRIACAO, CODIGONIVEL) VALUES (6, 'Computação Aplicada','10/09/1999', NULL);

```

O script utiliza três comandos: CREATE, ALTER e INSERT. A sintaxe completa desses comandos no PostgreSQL está disponível na seção Explore + ao final do conteúdo.

A imagem abaixo exibe o conteúdo da tabela NIVEL após a execução do comando `SELECT * FROM NIVEL`.

	codigonivel	descricao
1	1	Graduação
2	2	Especialização
3	3	Mestrado
4	4	Doutorado

A imagem a seguir apresenta o conteúdo da tabela CURSO após a execução do comando `TABLE CURSO;`

	codigocurso	nome	datacriacao	codigonivel
1	1	Sistemas de Informação	1999-06-19	1
2	2	Medicina	1990-05-10	1
3	3	Nutrição	2012-02-19	[NULL]
4	4	Pedagogia	1999-06-19	1
5	5	Saúde da Família	1999-09-10	3
6	6	Computação Aplicada	1999-09-10	[NULL]

As tabelas **NIVEL** e **CURSO** estão relacionadas, sendo **NIVEL** independente por não possuir chave estrangeira. Isso permite o cadastro de níveis sem depender de outras tabelas.

O relacionamento entre elas ocorre na tabela **CURSO**, onde a coluna **CODIGONIVEL** funciona como chave estrangeira. Como essa coluna é opcional, um curso pode ser cadastrado sem estar vinculado a um nível, como ocorreu nas inserções das linhas 20 e 23, onde **CODIGONIVEL** recebeu **NULL**.

A seguir, veremos o impacto de usar múltiplas tabelas em consultas SQL.

Operação de produto cartesiano

A tabela **NIVEL** possui duas colunas e quatro registros, enquanto a tabela **CURSO** tem quatro colunas e seis registros, conforme indicado no script anterior.

Ao executar a consulta `SELECT * FROM CURSO, NIVEL;`, obtemos o produto cartesiano dessas tabelas, cujo resultado está apresentado na imagem a seguir.

	codigocurso	nome	datacriacao	codigonivel	codigonivel	descricao
1	1	Sistemas de Informação	1999-06-19	1	1	Graduação
2	1	Sistemas de Informação	1999-06-19	1	2	Especialização
3	1	Sistemas de Informação	1999-06-19	1	3	Mestrado
4	1	Sistemas de Informação	1999-06-19	1	4	Doutorado
5	2	Medicina	1990-05-10	1	1	Graduação
6	2	Medicina	1990-05-10	1	2	Especialização
7	2	Medicina	1990-05-10	1	3	Mestrado
8	2	Medicina	1990-05-10	1	4	Doutorado
9	3	Nutrição	2012-02-19	[NULL]	1	Graduação
10	3	Nutrição	2012-02-19	[NULL]	2	Especialização
11	3	Nutrição	2012-02-19	[NULL]	3	Mestrado
12	3	Nutrição	2012-02-19	[NULL]	4	Doutorado
13	4	Pedagogia	1999-06-19	1	1	Graduação
14	4	Pedagogia	1999-06-19	1	2	Especialização
15	4	Pedagogia	1999-06-19	1	3	Mestrado
16	4	Pedagogia	1999-06-19	1	4	Doutorado
17	5	Saúde da Família	1999-09-10	3	1	Graduação
18	5	Saúde da Família	1999-09-10	3	2	Especialização
19	5	Saúde da Família	1999-09-10	3	3	Mestrado
20	5	Saúde da Família	1999-09-10	3	4	Doutorado
21	6	Computação Aplicada	1999-09-10	[NULL]	1	Graduação
22	6	Computação Aplicada	1999-09-10	[NULL]	2	Especialização
23	6	Computação Aplicada	1999-09-10	[NULL]	3	Mestrado
24	6	Computação Aplicada	1999-09-10	[NULL]	4	Doutorado

A consulta gerou uma tabela em memória a partir do **produto cartesiano**, combinando cada linha de **CURSO** com cada registro de **NIVEL**.

- A tabela resultante tem **6 colunas** (soma das colunas das tabelas envolvidas) e **24 registros** (multiplicação do número de linhas de **CURSO** e **NIVEL**).
- Enquanto cada linha de **NIVEL** ou **CURSO** representa um fato real do banco de dados, a tabela resultante não segue essa lógica, pois contém todas as combinações possíveis entre as linhas das duas tabelas.

Cuidado ao usar o produto cartesiano:

Esse tipo de consulta pode gerar um número excessivo de registros e sobrecarregar o servidor. Por exemplo, cruzar **1.000.000 de clientes** com **10.000.000 de pedidos** resultaria em **10 trilhões de linhas**, tornando a consulta inviável.

A mesma operação pode ser escrita como `SELECT * FROM CURSO CROSS JOIN NIVEL;`. A seguir, veremos como extrair informações úteis desse resultado.

Junção interna

A **junção interna (INNER JOIN)** permite combinar registros de duas tabelas com base em uma condição específica, retornando apenas os registros que possuem correspondência em ambas.

Essa técnica é essencial para extrair informações relevantes e otimizar consultas em bancos de dados relacionais. Por exemplo, ao cruzar as tabelas **NIVEL** e **CURSO**, identificamos que o curso **Sistemas de Informação** pertence ao nível de **graduação**, com base nos registros inseridos no banco de dados.

	<code>codigocurso</code>	<code>nome</code>	<code>datacriacao</code>	<code>codigonivel</code>	<code>codigonivel</code>	<code>descricao</code>
1	1	Sistemas de Informação	1999-06-19	1	1	Graduação
2	1	Sistemas de Informação	1999-06-19	1	2	Especialização
3	1	Sistemas de Informação	1999-06-19	1	3	Mestrado
4	1	Sistemas de Informação	1999-06-19	1	4	Doutorado

Apenas a primeira linha da consulta reflete a realidade do banco de dados, pois os valores da coluna **CODIGONIVEL** coincidem. As demais contêm dados inconsistentes.

Para eliminar essas linhas falsas, basta adicionar uma condição de igualdade entre as colunas correspondentes. As quatro primeiras colunas pertencem à tabela **CURSO**, enquanto as duas últimas vêm da tabela **NIVEL**.

```
SQL (SQLite 3.27.2)
1 SELECT *
2 FROM CURSO, NIVEL
3 WHERE NIVEL.CODIGONIVEL=CURSO.CODIGONIVEL;
```

	<code>codigocurso</code>	<code>nome</code>	<code>datacriacao</code>	<code>codigonivel</code>	<code>codigonivel</code>	<code>descricao</code>
1	1	Sistemas de Informação	1999-06-19	1	1	Graduação
2	2	Medicina	1990-05-10	1	1	Graduação
3	4	Pedagogia	1999-06-19	1	1	Graduação
4	5	Saúde da Família	1999-09-10	3	3	Mestrado

Foram recuperadas apenas as linhas que relacionam cursos aos seus níveis corretamente. A forma mais comum de obter esse resultado é usando a **junção interna (INNER JOIN)**, cuja sintaxe básica é:

```
SELECT *
FROM TABELA1 [INNER] JOIN TABELA2 ON (CONDIÇÃO_JUNÇÃO) [USING (COLUNA_DE_JUNÇÃO)]
```

Nessa sintaxe, a cláusula **JOIN** declara a junção entre as tabelas, seguida da preposição **ON** e da condição de junção, geralmente baseada na igualdade entre a chave primária de uma tabela e a chave estrangeira correspondente.

```
SQL (SQLite 3.27.2)
1 SELECT *
2 FROM CURSO INNER JOIN NIVEL ON(NIVEL.CODIGONIVEL=CURSO.CODIGONIVEL);
```

Note também que é possível declarar a cláusula **USING** especificando a coluna-alvo da junção. Em nosso exemplo, a coluna **CODIGONIVEL**. A consulta pode ser reescrita da seguinte forma:

```
SQL (SQLite 3.27.2)
1 SELECT *
2 FROM CURSO INNER JOIN NIVEL USING(CODIGONIVEL);
```

Se desejarmos exibir o código e o nome do curso, além do código e o nome do nível, podemos, então, executar o código a seguir:

```
SQL (SQLite 3.27.2)
1 SELECT CURSO.CODIGOCURSO, CURSO.NOME,
2 FROM CURSO INNER JOIN NIVEL USING(CODIGONIVEL);
```

No comando **SELECT**, a referência **NOMETABELA.NOMECOLUNA** é utilizada para especificar de qual tabela o valor será buscado. Isso é obrigatório para colunas como **CODIGONIVEL**, mas, por organização, pode ser aplicado a todas as colunas.

Além disso, para tornar o código mais claro e legível, é possível renomear as tabelas na consulta, facilitando sua compreensão e manutenção.

```
SQL (SQLite 3.27.2)
1 SELECT C.CODIGOCURSO, C.NOME,
2 N.CODIGONIVEL, N.DESCRICAO
3 FROM CURSO C INNER JOIN NIVEL N ON
4 (N.CODIGONIVEL=C.CODIGONIVEL) ;
```

	123 codigocurso	abc nome		123 codigonivel	abc descricao
1		1 Sistemas de Informação		1	Graduação
2		2 Medicina		1	Graduação
3		4 Pedagogia		1	Graduação
4		5 Saúde da Família		3	Mestrado

O resultado de uma junção interna corresponde somente aos registros que atendem à condição da junção, ou seja, os registros que de fato estão relacionados no contexto das tabelas envolvidas.

Junção externa

A junção externa no SQL permite combinar registros de duas tabelas, retornando não apenas os registros com correspondência, mas também os que não possuem. Usando a cláusula **LEFT JOIN** (junção à esquerda), é possível incluir todos os registros da tabela à esquerda, mesmo que não haja correspondência na tabela da direita. Quando não há correspondência, os resultados são preenchidos com **NULL**. Essa abordagem é útil para garantir que todos os dados de uma tabela sejam exibidos, independentemente da presença de registros correspondentes na outra tabela.

```
SQL (SQLite 3.27.2)
1 SELECT C.CODIGOCURSO, C.NOME,
2 N.CODIGONIVEL, N.DESCRICAO
3 FROM CURSO C LEFT JOIN NIVEL N ON (N.CODIGONIVEL=C.CODIGONIVEL) ;
```

	123 codigocurso	abc nome		123 codigonivel	abc descricao
1		1 Sistemas de Informação		1	Graduação
2		2 Medicina		1	Graduação
3		3 Nutrição		[NULL]	[NULL]
4		4 Pedagogia		1	Graduação
5		5 Saúde da Família		3	Mestrado
6		6 Computação Aplicada		[NULL]	[NULL]

O número de linhas do resultado da consulta corresponde ao número de registros da tabela CURSO, pois todos os registros dessa tabela estão incluídos no resultado. Cursos sem nível associado aparecem com valores **NULL**. A **RIGHT JOIN** (junção à direita) inclui todos os registros da tabela à direita, mesmo sem correspondência na tabela à esquerda, retornando **NULL** para os campos sem correspondência.

```
SQL (SQLite 3.27.2)
1 SELECT C.CODIGOCURSO, C.NOME,
2 N.CODIGONIVEL, N.DESCRICAO
3 FROM CURSO C RIGHT JOIN NIVEL N ON
4 (N.CODIGONIVEL=C.CODIGONIVEL);
```

	codigocurso	nome	codigonivel	descricao
1	1	Sistemas de Informação	1	Graduação
2	2	Medicina	1	Graduação
3	3	Nutrição	[NULL]	[NULL]
4	4	Pedagogia	1	Graduação
5	5	Saúde da Família	3	Mestrado
6	6	Computação Aplicada	[NULL]	[NULL]
7	[NULL]	[NULL]	2	Especialização
8	[NULL]	[NULL]	4	Doutorado

Todos os registros da tabela NIVEL aparecem no resultado. As quatro primeiras linhas correspondem aos registros relacionados à tabela CURSO, e as duas últimas linhas são registros sem relação com cursos no banco de dados. O mesmo resultado pode ser obtido usando junção à esquerda ou à direita, alternando as tabelas. Ou seja, TABELA1 LEFT JOIN TABELA2 é equivalente a TABELA2 RIGHT JOIN TABELA1.

```
SQL (SQLite 3.27.2)
1 SELECT C.CODIGOCURSO, C.NOME,
2 N.CODIGONIVEL, N.DESCRICAO
3 FROM CURSO C LEFT JOIN NIVEL N ON (N.CODIGONIVEL=C.CODIGONIVEL);
```

```
SQL (SQLite 3.27.2)
SELECT C.CODIGOCURSO, C.NOME,
       N.CODIGONIVEL, N.DESCRICAO
  FROM NIVEL N RIGHT JOIN CURSO C ON (N.CODIGONIVEL=C.CODIGONIVEL);
```

A junção FULL OUTER JOIN exibe todos os registros das tabelas à esquerda e à direita, incluindo os não relacionados. A tabela resultante mostra todos os registros de ambas as tabelas, com valores NULL para os registros sem correspondência.

```
SQL (SQLite 3.27.2)
1 SELECT C.CODIGOCURSO, C.NOME,
2 N.CODIGONIVEL, N.DESCRICAO
3 FROM CURSO C FULL OUTER JOIN NIVEL N ON
4 (N.CODIGONIVEL=C.CODIGONIVEL);
```

No resultado, aparecem os registros de cada tabela, com valores NULL nos casos sem correspondência (linhas 3, 6, 7 e 8). Quando é necessário extrair informações de mais de uma tabela, usamos a cláusula de junção. Os diferentes tipos de junção são escolhidos conforme a necessidade do usuário, pois cada um tem uma especificidade.

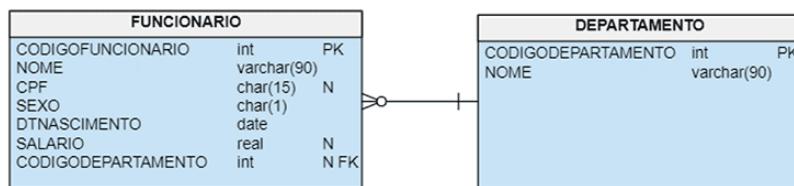
As **subconsultas aninhadas** permitem estruturar consultas complexas para obter informações detalhadas dos dados. Elas são úteis para realizar análises profundas e resolver problemas sofisticados de manipulação de dados. No vídeo, você aprenderá como aplicar essas consultas internas para realizar operações complexas no modelo relacional.

Subconsultas

O resultado de uma consulta SQL é uma tabela, mesmo que temporariamente armazenada na memória. Subconsultas são consultas que dependem dos resultados de outras consultas para recuperar informações. Existem dois tipos principais de subconsultas:

- **Subconsultas aninhadas:** A consulta externa realiza testes com os dados obtidos pela consulta interna.
- **Subconsultas correlatas:** A subconsulta utiliza valores da consulta externa e é executada para cada linha da consulta externa.

As consultas serão construídas com base nas tabelas FUNCIONARIO e DEPARTAMENTO. Para praticar, recomenda-se criar as tabelas e inserir dados usando um script no PostgreSQL.



A partir das tabelas FUNCIONARIO e DEPARTAMENTO, segue o script:

```
CREATE TABLE DEPARTAMENTO (
    CODIGODEPARTAMENTO int NOT NULL,
    NOME varchar(90) NOT NULL,
    CONSTRAINT DEPARTAMENTO_pk PRIMARY KEY (CODIGODEPARTAMENTO);
CREATE TABLE FUNCIONARIO (
    CODIGOFUNCIONARIO int NOT NULL,
    NOME varchar(90) NOT NULL,
    CPF char(15) NULL,
    SEXO char(1) NOT NULL,
    DTNASCIMENTO date NOT NULL,
    SALARIO real NULL,
    CODIGODEPARTAMENTO int NULL,
    CONSTRAINT FUNCIONARIO_pk PRIMARY KEY (CODIGOFUNCIONARIO));
ALTER TABLE FUNCIONARIO ADD CONSTRAINT FUNCIONARIO_DEPARTAMENTO
    FOREIGN KEY (CODIGODEPARTAMENTO) REFERENCES DEPARTAMENTO (CODIGODEPARTAMENTO);
INSERT INTO DEPARTAMENTO(CODIGODEPARTAMENTO,NOME) VALUES (1,'Tecnologia da Informação');
INSERT INTO DEPARTAMENTO(CODIGODEPARTAMENTO,NOME) VALUES (2,'Contabilidade');
INSERT INTO DEPARTAMENTO(CODIGODEPARTAMENTO,NOME) VALUES (3,'Marketing');
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO, CODIGODEPARTAMENTO)
VALUES (1, 'ROBERTA SILVA BRASIL',NULL, 'F', '20/02/1980',7000,1);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO, CODIGODEPARTAMENTO)
VALUES (2, 'MARIA SILVA BRASIL',NULL, 'F', '20/09/1988',9500,2);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO, CODIGODEPARTAMENTO)
VALUES (3,'GABRIELLA PEREIRA LIMA',NULL, 'F', '20/02/1990',6000,1);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO, CODIGODEPARTAMENTO)
VALUES (4, 'MARCOS PEREIRA BRASIL',NULL, 'M', '20/02/1999',6000,2);
INSERT INTO FUNCIONARIO (CODIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO, CODIGODEPARTAMENTO)
VALUES (5, 'HEMERSON SILVA BRASIL', NULL, 'M', '20/12/1992',4000,NULL);
```

A imagem a seguir apresenta o conteúdo da tabela DEPARTAMENTO após a execução do comando:
SELECT * FROM DEPARTAMENTO;

	codigodepartamento	nome
1	1	Tecnologia da Informação
2	2	Contabilidade
3	3	Marketing

A próxima imagem apresenta o conteúdo da tabela FUNCIONARIO após a execução do comando:
TABLE FUNCIONARIO; (equivalente a SELECT * FROM FUNCIONARIO;)

	codigofuncionario	nome	cpf	sexo	dtnascimento	salario	codigodepartamento
1	1	ROBERTA SILVA BRASIL	[NULL]	F	1980-02-20	7.000	1
2	2	MARIA SILVA BRASIL	[NULL]	F	1988-09-20	9.500	2
3	3	GABRIELLA PEREIRA LIMA	[NULL]	F	1990-02-20	6.000	1
4	4	MARCOS PEREIRA BRASIL	[NULL]	M	1999-02-20	6.000	2
5	5	HEMERSON SILVA BRASIL	[NULL]	M	1992-12-20	4.000	[NULL]

Perceba que, originalmente, existe um relacionamento do tipo 1:N entre DEPARTAMENTO e FUNCIONARIO. A implementação desse relacionamento ocorre por meio da chave estrangeira CODIGODEPARTAMENTO da tabela FUNCIONARIO.

Uma **subconsulta aninhada** é usada quando se precisa obter dados que dependem de uma ou mais consultas internas. Para isso, cria-se uma condição na cláusula WHERE, envolvendo o resultado da subconsulta em um teste.

Exemplo de consulta: Retornar o código e o nome do(s) funcionário(s) que ganham o maior salário.

```
SQL (SQLite 3.27.2)
SELECT CODIGOFUNCIONARIO, NOME
FROM FUNCIONARIO
WHERE SALARIO=
    (SELECT MAX(SALARIO)
     FROM FUNCIONARIO);
```

	codigofuncionario	nome
1	2	MARIA SILVA BRASIL

O SGBD processa primeiro a subconsulta, que retorna o maior salário registrado na tabela FUNCIONARIO. Esse valor é então utilizado na cláusula WHERE para filtrar os funcionários cujo salário satisfaz a condição especificada.

Exemplo de consulta: Retornar o código, o nome e o salário do(s) funcionário(s) que ganham mais que a média salarial dos colaboradores.

```
SQL (SQLite 3.27.2)
SELECT CODIGOFUNCIONARIO, NOME, SALARIO
FROM FUNCIONARIO
WHERE SALARIO>
    (SELECT AVG(SALARIO)
     FROM FUNCIONARIO);
```

	codigofuncionario	nome	salario
1	1	ROBERTA SILVA BRASIL	7.000
2	2	MARIA SILVA BRASIL	9.500

O SGBD processa primeiro a subconsulta, que retorna a média salarial a partir da tabela FUNCIONARIO. Esse valor é utilizado na cláusula WHERE para filtrar os funcionários cujo salário satisfaz a condição.

Exemplo de consulta: Retornar o código, o nome e o salário do(s) funcionário(s) que ganham menos que a média salarial dos colaboradores do departamento de Tecnologia da Informação (TI).

```
SQL (SQLite 3.27.2)
SELECT CODIGOFUNCIONARIO, NOME, SALARIO
FROM FUNCIONARIO
WHERE SALARIO<
    (SELECT AVG(SALARIO)
     FROM FUNCIONARIO
     WHERE CODIGODEPARTAMENTO IN (SELECT CODIGODEPARTAMENTO
                                    FROM DEPARTAMENTO
                                    WHERE NOME='Tecnologia da Informação'));
```

	123 codigofuncionario	abc nome	123 salario
1	3	GABRIELLA PEREIRA LIMA	6.000
2	4	MARCOS PEREIRA BRASIL	6.000
3	5	HEMERSON SILVA BRASIL	4.000

Para retornar os resultados de interesse, o SGBD precisa calcular a média salarial dos funcionários do departamento de TI. Para isso, a subconsulta da linha 4 calcula a média e utiliza o resultado da subconsulta da linha 6, que recupera o código do departamento.

Há uma alternativa para resolver a consulta 03, substituindo a subconsulta por uma junção. O código a seguir gera os mesmos resultados.

```
SQL (SQLite 3.27.2)
SELECT CODIGOFUNCIONARIO, NOME, SALARIO
FROM FUNCIONARIO
WHERE SALARIO>
    (SELECT AVG(SALARIO)
     FROM FUNCIONARIO F JOIN DEPARTAMENTO D ON
     (F.CODIGODEPARTAMENTO=D.CODIGODEPARTAMENTO)
     WHERE D.NOME='Tecnologia da Informação');
```

Consulta 04: quantos funcionários recebem menos que a funcionária que possui o maior salário entre as colaboradoras de sexo feminino?

```
SQL (SQLite 3.27.2)
SELECT COUNT(*) AS QUANTIDADE
FROM FUNCIONARIO
WHERE SALARIO<
    (SELECT MAX(SALARIO)
     FROM FUNCIONARIO
     WHERE SEXO='F');
```

	123 quantidade
1	4

Subconsultas correlatas são um tipo especial de subconsulta aninhada, onde os dados da consulta externa dependem do resultado da consulta interna. A condição na cláusula WHERE envolve o resultado da subconsulta para realizar um teste. Essas consultas são usadas para obter informações mais precisas e resolver problemas complexos de análise de dados de forma eficiente.

Consulta 05: Retornar o código, o nome e o salário do(s) funcionário(s) que ganha(m) mais que a média salarial dos colaboradores do departamento ao qual pertencem.

```
SQL (SQLite 3.27.2)
SELECT CODIGOFUNCIONARIO, NOME, SALARIO
FROM FUNCIONARIO F
WHERE SALARIO>
    (SELECT AVG(SALARIO)
     FROM FUNCIONARIO
     WHERE CODIGODEPARTAMENTO=F.CODIGODEPARTAMENTO);
```

	123 codigofuncionario	abc nome	123 salario
1	1	ROBERTA SILVA BRASIL	7.000
2	2	MARIA SILVA BRASIL	9.500

A consulta 05 é semelhante à consulta 02, mas a média salarial é calculada apenas para os funcionários de cada departamento, devido à condição WHERE na linha 6.

Há uma alternativa para resolver essa consulta, substituindo a subconsulta por uma junção. O código a seguir gera os mesmos resultados.

```
SQL (SQLite 3.27.2)

SELECT CODIGOFUNCIONARIO, NOME, SALARIO
FROM FUNCIONARIO F
JOIN
    (SELECT CODIGODEPARTAMENTO, AVG(SALARIO) AS MEDIA
     FROM FUNCIONARIO
     GROUP BY CODIGODEPARTAMENTO) TESTE
ON F.CODIGODEPARTAMENTO=TESTE.CODIGODEPARTAMENTO
WHERE SALARIO>MEDIA;
```

O exemplo a seguir demonstra o uso de uma consulta correlacionada em uma operação de atualização (UPDATE).

Consulta 06: Quando há a necessidade de equiparar os salários dos funcionários do mesmo departamento, os salários serão atualizados com base no maior salário de cada setor.

A imagem a seguir mostra os salários "antes" da atualização.

	codigofuncionario	nec nome	nec cpf	nec sexo	dtnascimento	salario	codigodepartamento
1	1	ROBERTA SILVA BRASIL	[NULL]	F	1980-02-20	7.000	1
2	3	GABRIELLA PEREIRA LIMA	[NULL]	F	1990-02-20	6.000	1
3	2	MARIA SILVA BRASIL	[NULL]	F	1988-09-20	9.500	2
4	4	MARCOS PEREIRA BRASIL	[NULL]	M	1999-02-20	6.000	2
5	5	HEMERSON SILVA BRASIL	[NULL]	M	1992-12-20	4.000	[NULL]

Perceba que a listagem está ordenada pela coluna CODIGODEPARTAMENTO. O maior salário de funcionário pertencente ao departamento 1 é R\$ 7.000; em relação ao departamento 2, R\$ 9.500. Note também que há um funcionário sem a informação sobre departamento.

```
SQL (SQLite 3.27.2)

UPDATE FUNCIONARIO F
SET SALARIO=
    (SELECT MAX(SALARIO)
     FROM FUNCIONARIO
     WHERE CODIGODEPARTAMENTO=F.CODIGODEPARTAMENTO)
WHERE F.CODIGODEPARTAMENTO IS NOT NULL;
```

	codigofuncionario	nec nome	nec cpf	nec sexo	dtnascimento	salario	codigodepartamento
1	1	ROBERTA SILVA BRASIL	[NULL]	F	1980-02-20	7.000	1
2	3	GABRIELLA PEREIRA LIMA	[NULL]	F	1990-02-20	7.000	1
3	2	MARIA SILVA BRASIL	[NULL]	F	1988-09-20	9.500	2
4	4	MARCOS PEREIRA BRASIL	[NULL]	M	1999-02-20	9.500	2
5	5	HEMERSON SILVA BRASIL	[NULL]	M	1992-12-20	4.000	[NULL]

A consulta tem o objetivo de recuperar o maior salário de cada departamento e usar esse valor para atualizar os salários dos funcionários na tabela FUNCIONARIO, conforme o departamento de cada colaborador. A atualização ocorre apenas para os funcionários que têm um departamento associado, devido à cláusula WHERE na linha 6, que impede a atualização de salários de funcionários sem departamento.

A sintaxe completa do comando UPDATE no PostgreSQL está disponível na seção Explore + ao final do material.

Consulta correlacionada com uso de [NOT] EXISTS

A consulta correlacionada com o operador [NOT] EXISTS é usada para testar a existência de registros retornados por uma subconsulta. O operador EXISTS verifica se a subconsulta retorna linhas, enquanto o [NOT] EXISTS verifica se não há registros. Essa técnica é útil em consultas complexas para garantir resultados precisos nas análises de dados.

Consulta 07: exibir o código e o nome do departamento em que há pelo menos um funcionário alocado.

SQL (SQLite 3.27.2)	
	SELECT D.CODIGODEPARTAMENTO, D.NOME FROM DEPARTAMENTO D WHERE EXISTS (SELECT F.CODIGODEPARTAMENTO FROM FUNCIONARIO F WHERE D.CODIGODEPARTAMENTO=F.CODIGODEPARTAMENTO);
123	codigodepartamento ABC nome

	123 codigodepartamento	ABC nome
1		Tecnologia da Informação
2		Contabilidade

A subconsulta correlacionada (linhas 4 a 6) é executada, e se retornar pelo menos uma linha, a cláusula WHERE retorna verdadeiro e as colunas especificadas na linha 1 são exibidas.

Se quisermos verificar se há algum departamento sem colaborador alocado, podemos usar a negação (NOT) na consulta, como mostrado a seguir.

SQL (SQLite 3.27.2)	
	SELECT D.CODIGODEPARTAMENTO, D.NOME FROM DEPARTAMENTO D WHERE NOT EXISTS (SELECT F.CODIGODEPARTAMENTO FROM FUNCIONARIO F WHERE D.CODIGODEPARTAMENTO=F.CODIGODEPARTAMENTO);
123	codigodepartamento ABC nome

	123 codigodepartamento	ABC nome
1		Marketing

Operadores de conjunto

Os operadores de conjunto no SQL permitem combinar, interseccionar e diferenciar conjuntos de dados de forma eficaz. Os principais operadores são UNION, INTERSECT e EXCEPT, que combinam os resultados de diversas consultas em um único conjunto de dados, desde que sigam regras específicas.

FUNCIONARIO		
CODIGO FUNCIONARIO	int	PK
NOME	varchar(90)	
CPF	char(15)	N
SEXO	char(1)	
DTNASCIMENTO	date	
SALARIO	real	N

ALUNO		
CODIGO ALUNO	int	PK
NOME	varchar(90)	
CPF	char(15)	
SEXO	char(1)	
DTNASCIMENTO	date	

CLIENTE		
CODIGO CLIENTE	int	PK
NOME	varchar(90)	
CPF	char(15)	
SEXO	char(1)	

Recomenda-se criar as tabelas e inserir dados usando o script fornecido, a partir da ferramenta de sua preferência. Lembre-se de estar conectado ao PostgreSQL e acessar um banco de dados criado por você.

```
CREATE TABLE FUNCIONARIO (
    CODIGO FUNCIONARIO int NOT NULL,
    NOME varchar(90) NOT NULL,
    CPF char(15) NULL,
    SEXO char(1) NOT NULL,
    DTNASCIMENTO date NOT NULL,
    SALARIO real NULL,
    CONSTRAINT FUNCIONARIO_pk PRIMARY KEY (CODIGO FUNCIONARIO));
CREATE TABLE ALUNO (
    CODIGO ALUNO int NOT NULL,
    NOME varchar(90) NOT NULL,
    CPF char(15) NOT NULL,
    SEXO char(1) NOT NULL,
    DTNASCIMENTO date NOT NULL,
    CONSTRAINT ALUNO_pk PRIMARY KEY (CODIGO ALUNO));
CREATE TABLE CLIENTE (
    CODIGO CLIENTE int NOT NULL,
    NOME varchar(90) NOT NULL,
    CPF char(15) NOT NULL,
    SEXO char(1) NOT NULL,
    CONSTRAINT CLIENTE_pk PRIMARY KEY (CODIGO CLIENTE));
```

O script a seguir pode ser utilizado para inserção de registros nas tabelas.

```
INSERT INTO FUNCIOILARIO (COOIGOFUNCIONARIO, NOME, CPF, SEXO, DTMASCIMENTO, SALARIO)
VALUES (1, 'ROBERTA SILVA BRASIL', '82998', 'F', '20/02/1980', 70e0);
INSERT INTO FUNCIOILARIO (COOIGOFUNCIONARIO, NOME, CPF, SEXO, DTMASCIMENTO, SALARIO)
VALUES (2, 'MARIA SILVA BRASIL', '9876', 'F', '20/09/1988', 9500);
INSERT INTO FUNCIOIARIO (COOIGOFUNCIONARIO, NOME, CPF, SEXO, DTNASCIMENTO, SALARIO)
VALUES (3, 'GABRIELLA PEREIRA LIMA', '32998', 'F', '20/02/1990', 6000);
INSERT INTO FUNCIOHLARIO (COOIGOFUNCIONARIO, NOME, CPF, SEXO, DTMASCIMENTO, SALARIO)
VALUES (4, 'MARCOS PEREIRA BRASIL', '9999', 'M', '20/02/1999', 60e0);
INSERT INTO FUNCIOHLARIO (COOIGOFUNCIONARIO, NOME, CPF, SEXO, DTMASCIMENTO, SALARIO)
VALUES (5, 'HEMERSUI SILVA BRASIL', '9111', 'H', '20/12/1992', 40e0);
INSERT INTO ALUNO (CODOALUNO, NOME, CPF, SEXO, DTNASCIMENTO) VALUES (1, 'JOSÉ FRANCISCO TERRA', '82988', 'M', '28/10/1989');
INSERT INTO ALUNO (CODOALUNO, NOME, CPF, SEXO, DTNASCIMENTO) VALUES (2, 'ANDREY COSTA FILHO', '0024', 'M', '20/10/1999');
INSERT INTO ALUNO (CODOALUNO, NOME, CPF, SEXO, DTNASCIMENTO) VALUES (3, 'ROBERTA SILVA BRASIL', '82998', 'F', '20/02/1980');
INSERT INTO ALUNO (CODOALUNO, NOME, CPF, SEXO, DTNASCIMENTO) VALUES (4, 'CARLA MARIA MACIEL', '0044', 'F', '20/11/1996');
INSERT INTO ALUNO (CODOALUNO, NOME, CPF, SEXO, DTNASCIMENTO) VALUES (5, 'MARCOS PEREIRA BRASIL', '9999', 'M', '20/02/1999');
INSERT INTO CLIENTE (COOIGOCLEIMTE, NOME, CPF, SEXO) VALUES (1, 'ROBERTA SILVA BRASIL', '82998', 'F');
INSERT INTO CLIENTE (COOIGOCLEIMTE, NOME, CPF, SEXO) VALUES (2, 'MARCOS PEREIRA BRASIL', '9999', 'M');
20 INSERT INTO CLIENTE (COOIGOCLEIMTE, NOME, CPF, SEXO) VALUES (3, 'HEMERSON SILVA BRASIL', '9111', 'M');
```

A imagem a seguir apresenta o conteúdo da tabela FUNCIONARIO após a execução do comando:
SELECT * FROM FUNCIONARIO;

	<code>123</code> codigofuncionario	abc nome	abc cpf	abc sexo	dtnascimento	123 salario
1	1	ROBERTA SILVA BRASIL	82998	F	1980-02-20	7.000
2	2	MARIA SILVA BRASIL	9876	F	1988-09-20	9.500
3	3	GABRIELLA PEREIRA LIMA	32998	F	1990-02-20	6.000
4	4	MARCOS PEREIRA BRASIL	9999	M	1999-02-20	6.000
5	5	HEMERSON SILVA BRASIL	9111	M	1992-12-20	4.000

A imagem a seguir apresenta o conteúdo da tabela ALUNO após a execução do comando: TABLE ALUNO;

	<code>123</code> codigoaluno	abc nome	abc cpf	abc sexo	dtnascimento
1	1	JOSÉ FRANCISCO TERRA	82988	M	1989-10-28
2	2	ANDREY COSTA FILHO	0024	M	1999-10-20
3	3	ROBERTA SILVA BRASIL	82998	F	1980-02-20
4	4	CARLA MARIA MACIEL	0044	F	1996-11-20
5	5	MARCOS PEREIRA BRASIL	9999	M	1999-02-20

A imagem a seguir apresenta o conteúdo da tabela CLIENTE após a execução do comando: TABLE CLIENTE;

	<code>123</code> codigocliente	abc nome	abc cpf	abc sexo
1	1	ROBERTA SILVA BRASIL	82998	F
2	2	MARCOS PEREIRA BRASIL	9999	M
3	3	HEMERSON SILVA BRASIL	9111	M

Os dados são fictícios, e os valores da coluna CPF possuem apenas quatro caracteres, não sendo um CPF válido. Para os efeitos do estudo, consideramos que registros com o mesmo CPF representam o mesmo cidadão. Além disso, cada tabela pertence a um banco de dados e domínio de aplicação diferentes.

Consultas com o operador UNION

O operador UNION é usado para consolidar dados de consultas diferentes em uma única tabela. As consultas envolvidas devem ter o mesmo número de colunas e tipos de dados compatíveis. O operador elimina duplicatas, retornando uma tabela sem linhas repetidas. A sintaxe geral é:

`CONSULTASQL UNION [ALL|DISTINCT] CONSULTASQL`

A sintaxe completa do comando SELECT no PostgreSQL está disponível na seção Explore + ao final do material.

Consulta 01: retornar o nome e o CPF de todos os funcionários e clientes.

	nome	cpf
1	ROBERTA SILVA BRASIL	82998
2	GABRIELLA PEREIRA LIMA	32998
3	MARCOS PEREIRA BRASIL	9999
4	MARIA SILVA BRASIL	9876
5	HEMERSON SILVA BRASIL	9111

Há cinco funcionários cadastrados (linhas 1 a 10) e três clientes (linhas 18 a 20), sendo que todos os clientes são também funcionários. Após a operação de união, apenas cinco registros são exibidos, pois duplicatas são eliminadas por padrão.

Para exibir todos os registros, incluindo as duplicatas, deve-se usar o **UNION ALL**, como mostrado a seguir.

	nome	cpf
1	ROBERTA SILVA BRASIL	82998
2	MARIA SILVA BRASIL	9876
3	GABRIELLA PEREIRA LIMA	32998
4	MARCOS PEREIRA BRASIL	9999
5	HEMERSON SILVA BRASIL	9111
6	ROBERTA SILVA BRASIL	82998
7	MARCOS PEREIRA BRASIL	9999
8	HEMERSON SILVA BRASIL	9111

Finalmente, se quiséssemos especificar a “origem” de cada registro, poderíamos alterar o nosso código conforme a seguir.

```
SQL (SQLite 3.27.2)
SELECT NOME, CPF, 'Dados da tabela FUNCIONARIO' AS ORIGEM
FROM FUNCIONARIO
UNION ALL
SELECT NOME, CPF, 'Dados da tabela CLIENTE' AS ORIGEM
FROM CLIENTE;
```

	nome	cpf	origem
1	ROBERTA SILVA BRASIL	82998	Dados da tabela FUNCIONARIO
2	MARIA SILVA BRASIL	9876	Dados da tabela FUNCIONARIO
3	GABRIELLA PEREIRA LIMA	32998	Dados da tabela FUNCIONARIO
4	MARCOS PEREIRA BRASIL	9999	Dados da tabela FUNCIONARIO
5	HEMERSON SILVA BRASIL	9111	Dados da tabela FUNCIONARIO
6	ROBERTA SILVA BRASIL	82998	Dados da tabela CLIENTE
7	MARCOS PEREIRA BRASIL	9999	Dados da tabela CLIENTE
8	HEMERSON SILVA BRASIL	9111	Dados da tabela CLIENTE

Consultas com os operadores INTERSECT e EXCEPT

Os operadores **INTERSECT** e **EXCEPT** são usados para comparar e manipular conjuntos de dados de diferentes fontes em uma única consulta.

INTERSECT exibe as linhas que aparecem em ambos os resultados das consultas, eliminando duplicatas. As consultas devem ter o mesmo número de colunas e tipos de dados compatíveis. A sintaxe geral é:

CONSULTASQL INTERSECT [ALL|DISTINCT] CONSULTASQL

Consulta 02: retornar o nome e o CPF de todos os cidadãos que são funcionários e clientes.

SQL (SQLite 3.27.2)		
	ABC nome	ABC cpf
1	MARCOS PEREIRA BRASIL	9999
2	ROBERTA SILVA BRASIL	82998
3	HEMERSON SILVA BRASIL	9111

A consulta retorna três linhas que são fruto da interseção entre as tabelas FUNCIONARIO e CLIENTE. Como visto, todos os clientes são funcionários.

Consulta 03: retornar o nome e o CPF de todos os cidadãos que são funcionários, clientes e alunos.

SQL (SQLite 3.27.2)		
	ABC nome	ABC cpf
1	MARCOS PEREIRA BRASIL	9999
2	ROBERTA SILVA BRASIL	82998

A consulta retorna duas linhas que são fruto da interseção entre as tabelas FUNCIONARIO, CLIENTE e ALUNO.

Um aspecto importante é que uma consulta sob o formato X UNION Y INTERSECT Z é interpretada sendo X UNION (Y INTERSECT Z).

SQL (SQLite 3.27.2)		
	ABC nome	ABC cpf
1	ROBERTA SILVA BRASIL	82998
2	GABRIELLA PEREIRA LIMA	32998
3	MARCOS PEREIRA BRASIL	9999
4	MARIA SILVA BRASIL	9876
5	HEMERSON SILVA BRASIL	9111

O operador **EXCEPT** realiza a operação de subtração dos conjuntos, exibindo linhas que aparecem em uma consulta e não na outra. As consultas devem ter o mesmo número de colunas e tipos de dados compatíveis, e duplicatas são eliminadas. A sintaxe geral é:

CONSULTASQL EXCEPT [ALL|DISTINCT] CONSULTASQL

Alguns SGBDs, como o Oracle, usam o operador **MINUS** para essa operação.

Consulta 04: retornar o nome e o CPF dos funcionários que não são clientes.

```
SQL (SQLite 3.27.2)
SELECT NOME, CPF
FROM FUNCIONARIO
EXCEPT
SELECT NOME, CPF
FROM CLIENTE;
```

	ABC nome	V↓	ABC cpf	V↓
1	GABRIELLA PEREIRA LIMA		32998	
2	MARIA SILVA BRASIL		9876	

A consulta retorna duas linhas que são fruto da subtração entre as tabelas FUNCIONARIO e CLIENTE.

Perceba que uma operação X EXCEPT Y é diferente de Y EXCEPT X.

```
SQL (SQLite 3.27.2)
SELECT NOME, CPF
FROM CLIENTE
EXCEPT
SELECT NOME, CPF
FROM FUNCIONARIO;
```

A consulta retorna vazio, pois todos os clientes são funcionários.

Consulta 05: retornar o nome e o CPF dos cidadãos que são somente funcionários.

```
SQL (SQLite 3.27.2)
SELECT NOME, CPF
FROM FUNCIONARIO
EXCEPT
SELECT NOME, CPF
FROM CLIENTE
EXCEPT
SELECT NOME, CPF
FROM ALUNO;
```

O resultado da consulta 05 está expresso na próxima imagem.

Inicialmente, o SGBD processa a operação de subtração da linha 3. Em seguida, o resultado da operação é usado na subtração da linha 6.

Vou enviar o conteúdo de uma página de uma apostila de estudo. Preciso de um resumo claro e direto. Eu não quero que você simplifique as coisas ao ponto de deixar o mínimo de informações possível, não é esse o objetivo aqui, o objetivo é pegar o texto original, e simplificá-lo, tirando quaisquer redundâncias, repetições, ou palavras desnecessárias. Se algo pode ser dito eficientemente com 5 palavras, não há necessidade de usar 10. Texto a ser resumido:

110+20+70+25+7,38+14,25

