Banco de Dados com *Python*

Seja MUITO bem-vindo(a), ao Curso de Banco de Dados com *Python*!

Estamos muito felizes de que tal tema o(a) tenha interessado. Saiba que nosso curso foi pensado com muito carinho, para que você acesse informações relevantes e de forma prática sobre essa importante e tão presente área da nossa sociedade.

Não se preocupe se no começo as coisas parecerem um pouco abstratas, e não desanime, pois ao longo dos conteúdos você perceberá que tudo fará sentido, e melhor, que você poderá treinar imediatamente fazendo uso dos exemplos dados de forma prática.

Então vamos lá, juntos aprender a utilizar essa importantíssima ferramenta para a manipulação de informações, a qual se tornou simplesmente primordial para a Sociedade do Conhecimento.

Atualmente o volume de dados tem atingindo proporções sem precedentes, e para tamanha massa de informações ser organizada, os bancos de dados desempenham um papel crucial, pois não apenas possibilitam armazenamento eficiente, mas também viabilizam acesso rápido de modo a tornar as mesmas em conhecimento.

A importância dos SGBDs - Sistemas Gerenciadores de Banco de Dados (*Data Base Management System*) é indiscutível nesse cenário. Essas ferramentas desempenham um papel vital ao permitir a organização, recuperação e manipulação de dados de forma estruturada e eficiente. Sem os SGBDs, a tarefa de gerenciar a vasta quantidade de informações seria monumental e propensa a erros.

A linguagem SQL - *Structured Query Language* ou Linguagem de Consulta Estruturada, é outra peça fundamental nesse quebra-cabeças. Ela proporciona uma maneira padronizada e poderosa de interagir com os bancos de dados. Por meio de comandos SQL, é possível realizar consultas, inserções, atualizações e exclusões de dados com facilidade, tornando a gestão de informações mais acessível a usuários com diferentes níveis de conhecimento técnico.

Aliado a isso, está a linguagem de programação *Python*, a qual por ser extremamente versátil, tem se destacado e sido amplamente adotada como uma escolha popular para acesso e análise de dados. Sua sintaxe clara e extensa gama de bibliotecas voltadas para manipulação de dados e análises estatísticas a tornam uma ferramenta valiosa para explorar os *insights* ocultos nos bancos de dados.

Este curso vai além da exploração dos conceitos fundamentais dos bancos de dados; ele reconhece o papel crítico desempenhado pelos SGBDs, a linguagem SQL e a linguagem *Python* como parceiros indispensáveis na gestão, análise e transformação dessas informações em conhecimento estratégico. Tais capacidades nos permitem navegar com confiança em um ambiente de dados cada vez mais complexo e valioso.

1 - CONCEITOS DE BANCO DE DADOS

No mundo contemporâneo, onde a informação flui em volumes sem precedentes, os conceitos de banco de dados desempenham um papel central. Eles constituem a base para a organização, armazenamento e gerenciamento eficaz de dados, transformando elementos brutos em informações com significado. Essas informações, por sua vez, são essenciais para a tomada de decisões informadas de forma assertiva.

Dados, Informações e Conhecimentos

Os **dados** são a matéria-prima inicial, consistindo em números, palavras e fatos sem contexto ou interpretação. Imagine um conjunto de números desconexos, como: 25, 37, 18, 42; esses são dados em sua forma mais rudimentar. No entanto, quando organizados e processados, esses dados se transformam em **informações**, ganhando contexto e utilidade. Por exemplo, ao associar esses números às idades de pessoas em uma pesquisa, eles se tornam informações relevantes, tais como: 25 anos, 37 anos, 18 anos, 42 anos.

Avançando além das informações, chegamos ao estágio dos **conhecimentos**. É nesse ponto que ocorre a interpretação das informações. Por exemplo, ao perceber que a média de idade dos participantes de uma pesquisa é maior do que o esperado, você pode interpretar essa informação como uma oportunidade para desenvolver produtos voltados a um público mais maduro. Esse é o conhecimento - a compreensão extraída das informações, possibilitando decisões informadas e estratégias conscientes.

Definição de Banco de Dados

Um banco de dados é um conjunto organizado de dados inter-relacionados, armazenados e gerenciados de forma eficaz para atender a diversas necessidades. Pode abranger desde informações sobre clientes em um estabelecimento até registros médicos em uma instituição de saúde.

Sistemas Gerenciadores de Banco de Dados (SGBDs)

Os Sistemas Gerenciadores de Banco de Dados (SGBDs) são *softwares* projetados para facilitar a criação, manipulação e administração de bancos de dados. Eles atuam como intermediários entre os usuários e os dados, fornecendo uma interface de acesso e executando operações de gerenciamento. Os SGBDs garantem a integridade, consistência e segurança dos dados armazenados. Alguns exemplos populares incluem *MySQL*, *PostgreSQL*, *Oracle* e *Microsoft SQL Server*.

Aplicações dos SGBDs

Os Sistemas Gerenciadores de Banco de Dados (SGBDs) encontram um amplo espectro de aplicabilidade em diversas esferas e setores. No contexto empresarial, eles desempenham um papel central na administração de dados relacionados a clientes, transações financeiras, estoques, recursos humanos e outras atividades operacionais. Na área da saúde, os SGBDs são empregados para armazenar com segurança registros médicos, resultados de exames e informações vitais dos pacientes.

Nos sistemas de comércio eletrônico, essas ferramentas gerenciam meticulosamente catálogos de produtos, processam pedidos e mantém informações cruciais sobre os clientes. Contudo, essa é apenas uma pequena amostra das possíveis aplicações, pois os SGBDs apresentam sua utilidade em virtualmente todos os âmbitos que demandam o

armazenamento e a recuperação de dados.

À medida que o mundo se volta para a automação e a digitalização, os SGBDs ganham ainda mais relevância. Os registros transacionais que mantêm são essenciais para operações eficientes e análises precisas. Eles não apenas facilitam a organização dos dados transacionais, mas também garantem sua integridade e disponibilidade. Em resumo, os SGBDs são os alicerces sobre os quais repousa uma ampla gama de atividades empresariais e operacionais modernas, proporcionando a base sólida para a tomada de decisões informadas e eficazes.

Vantagens dos SGBDs

- **Organização Estruturada:** Proporcionam uma estrutura organizada para armazenar e gerenciar dados, permitindo a definição de esquemas e relacionamentos entre as entidades. Isso facilita a visualização e compreensão dos dados, além de viabilizar a execução de consultas complexas.
- **Acesso Eficiente aos Dados:** São projetados para garantir um acesso rápido e eficiente aos dados armazenados. Utilizam índices, otimização de consultas e técnicas de armazenamento eficientes para reduzir o tempo necessário para recuperar informações específicas.
- **Controle de Segurança Avançado:** Oferecem recursos avançados de segurança para proteger os dados contra acesso não autorizado. Permitem a definição de permissões de acesso granulares, criptografia dados sensíveis e registro atividades de usuários para fins de auditoria.
- **Concorrência e Controle de Transações:** Possuem a capacidade de lidar com múltiplos usuários e transações concorrentes de forma consistente e segura. Garantem a Atomicidade, Consistência, Isolamento e Durabilidade (ACID) das transações, evitando problemas de inconsistência e corrupção dos dados.

Desvantagens dos SGBDs

- **Complexidade de Implementação e Administração:** Podem ser complexos de implementar e administrar. Exigem conhecimentos especializados e habilidades técnicas para garantir sua configuração adequada, desempenho otimizado e segurança eficaz.
- **Custo de Licenciamento e Infraestrutura:** Alguns SGBDs comerciais podem ter custos significativos de licenciamento, especialmente para empresas de pequeno porte. Além disso, a infraestrutura necessária para hospedar um SGBD, como servidores e armazenamento, também pode representar um investimento considerável.
- **Limitações de Escalabilidade:** Embora sejam escaláveis, podem enfrentar limitações em termos de dimensionamento horizontal para lidar com volumes extremamente grandes de dados ou cargas de trabalho intensivas. Em tais situações, soluções de banco de dados distribuídos ou sistemas *NoSQL* [1] podem ser mais adequados.

Sistemas de Banco de Dados

Um Sistema de Banco de Dados (SBD) é composto por uma coleção de informações interligadas e um Sistema de Gerenciamento de Banco de Dados (SGBD). O SGBD capacita os usuários a definir, construir e manipular bases de dados para diversas aplicações. Isso envolve a descrição dos tipos de dados, armazenamento, consulta e geração de relatórios.

Compreender esses conceitos é fundamental para explorar a fundo os SBDs, modelos de dados, SGBDs e a linguagem SQL ao longo de nosso curso. Esses conhecimentos capacitam a interação eficaz com dados em ambientes de banco de dados complexos e em constante evolução.

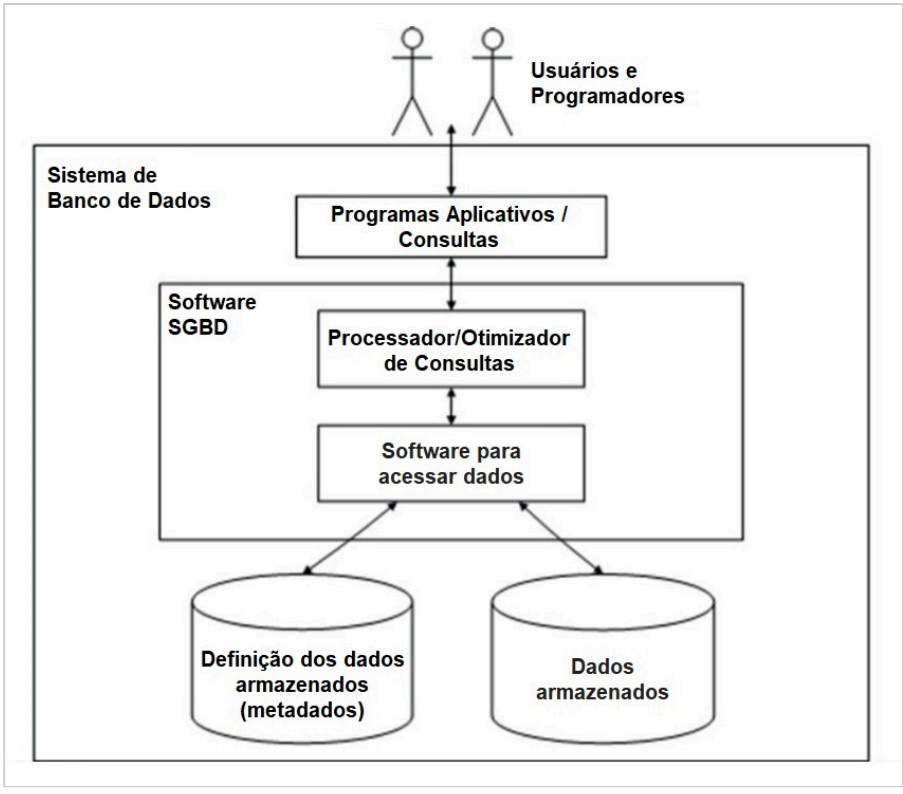

Figura 1: Diagrama simplificado de um ambiente de SBD (ELMASRI; NAVATHE, 2011)

A combinação da base de dados propriamente dita e do *software* de gerenciamento da base de dados compõe o que é conhecido como um Sistema de Base de Dados. Na Figura 1, é apresentada uma representação genérica de como ocorre a interação entre um Sistema de Banco de Dados e seus usuários.

---

[1] *Not Only SQL* (não só SQL) ou apenas *NoSQL*, refere-se a um termo genérico que representa os bancos de dados não relacionais, os quais são criados para modelos de dados específicos e têm esquemas flexíveis para a criação de aplicativos modernos. Entre os principais bancos desse tipo, temos: *MongoDB*, *Redis*, *Cassandra*, *Hbase* e *Amazon DynamoDB*.

2 - MODELAGEM DE DADOS

A Modelagem de Dados oferece uma abordagem sistemática para projetar bancos de dados que se alinham com as necessidades da aplicação e garantem a consistência, integridade e eficiência no armazenamento e recuperação de informações. Ela facilita a comunicação entre desenvolvedores, projetistas e usuários finais, resultando em sistemas de banco de dados bem projetados e capazes de lidar com os desafios do mundo real de maneira eficaz. Esse processo ocorre em diferentes fases, conforme Figura 2, cada uma delas contribuindo para a definição da estrutura e do relacionamento dos dados armazenados em um banco de dados.

O Modelo Conceitual é a primeira etapa da modelagem de dados, na qual são identificados e definidos os principais conceitos do mundo real que serão representados no banco de dados. Isso inclui a identificação de **entidades** (objetos, conceitos), **atributos** (propriedades) dessas **entidades** e **relacionamentos** entre as entidades. O objetivo é criar uma visão abstrata e independente de detalhes técnicos, focando na compreensão clara das informações que o sistema precisa capturar.

O Modelo Lógico é uma evolução do modelo conceitual, onde os conceitos abstratos são traduzidos em estruturas mais próximas do que será implementado em um SGBD. Nessa fase, as **entidades se transformam em tabelas**, os **atributos em coluna**s e os **relacionamentos em chaves primárias e estrangeiras**.

O Modelo Físico é a etapa final da modelagem de dados, em que a estrutura lógica definida no modelo lógico é mapeada para as especificações técnicas do SGBD escolhido. Isso inclui detalhes como **tipos de dados para cada coluna**, **criação de índices** para melhorar o desempenho de consultas, **definição de restrições** para garantir a integridade dos dados e **escolha de estratégias** de armazenamento para otimização do acesso aos dados.

Você Sabia?

A modelagem de dados não começou com os modelos relacionais que são amplamente utilizados hoje. No início, o modelo hierárquico (semelhante a uma árvore de diretórios/pastas) e o modelo de rede (que permitia múltiplas relações entre registros) dominavam a cena. Foi somente em 1970 que *Edgar Frank Codd*, um cientista da IBM, propôs o modelo relacional, que se tornou a base para a maioria dos SGBDs modernos. Este modelo foi revolucionário porque permitiu uma representação mais simples e flexível dos dados, além de proporcionar uma base teórica sólida (álgebra relacional) para manipulação de dados.

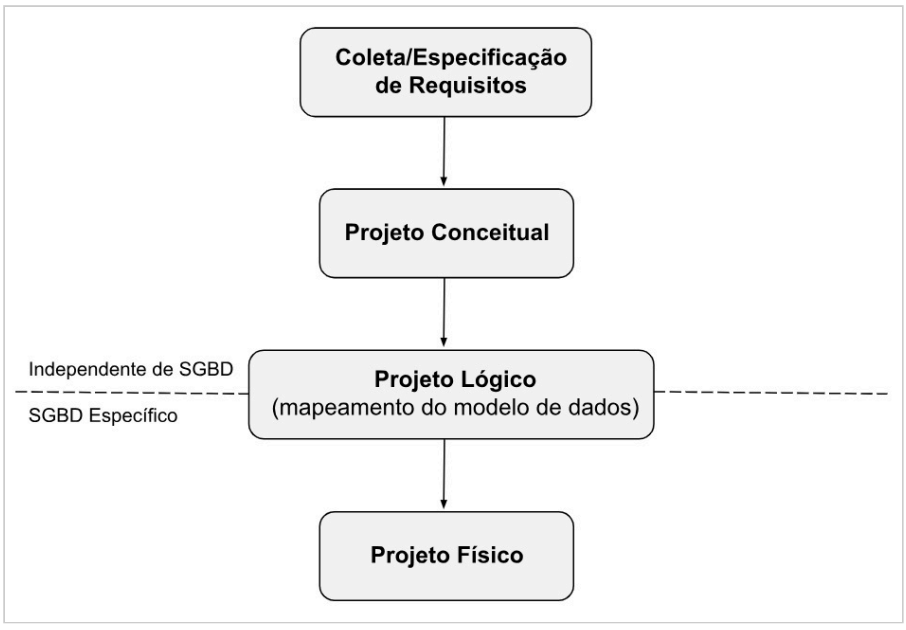

Figura 2: Diagrama simplificado de um ambiente de sistema de banco de dados - Adaptado de (ELMASRI; NAVATHE, 2011)

Modelo Conceitual

O processo de desenvolvimento de um banco de dados é iniciado com a criação de um modelo conceitual, uma etapa crucial que define as estruturas de alto nível e as relações entre os dados. Em sua essência, ele permite uma compreensão abstrata e simplificada do banco de dados, independentemente de detalhes de implementação. Auxilia os projetistas a capturar os requisitos essenciais de dados e a definir a estrutura lógica do sistema de forma clara e coerente.

Nesse contexto, o **Modelo Entidade-Relacionamento** (MER) ou simplesmente (ER), desenvolvido por *Peter Pin-Shan Chen* pesquisador do MIT <https://www.mit.edu/> em 1976, é uma das abordagens mais amplamente adotadas para a criação de modelos conceituais. Esse modelo fornece uma representação visual que permite descrever a estrutura de dados de um sistema ou domínio de aplicação de forma clara e abstrata. Uma das principais vantagens do modelo ER reside em sua capacidade de simplificar a complexidade dos dados, tornando-os mais acessíveis e compreensíveis.

O modelo ER descreve a estrutura do banco de dados, identificando as entidades envolvidas, seus atributos e os relacionamentos entre elas.

Você Sabia?

O modelo conceitual é muitas vezes comparado ao "esqueleto" ou "planta baixa" de um edifício. Assim como um arquiteto começa desenhando uma planta para visualizar a estrutura geral de um edifício antes de se preocupar com detalhes específicos de

construção, um *designer* de banco de dados começa com o modelo conceitual para obter uma visão geral dos dados e suas relações.

Entidades

As entidades são os objetos do mundo real que são representados no banco de dados. Podem ser pessoas, lugares, objetos, eventos, conceitos abstratos, etc. As instâncias de uma entidade não são representadas no diagrama, mas são semanticamente interpretadas no mesmo. O conjunto de entidades é representado por retângulos no diagrama ER, e seus nomes são escritos dentro de retângulos. Por exemplo, uma entidade pode ser "Cliente", representada por um retângulo com o nome **"Clientes"** dentro dele.

Figura 3: Representação da entidade **"Clientes"**

Atributos

Os atributos são características ou propriedades que descrevem as entidades. Representam informações específicas sobre as entidades e são usados para armazenar e manipular dados.

Os valores de um atributo que descrevem as entidades podem ser simples ou compostos, monovalorados ou multivalorados, nulos e derivados. Os atributos simples são indivisíveis e não podem ser subdivididos, como "Nome" ou "Cidade", e são representados no diagrama ER por elipses conectadas às entidades por linhas.

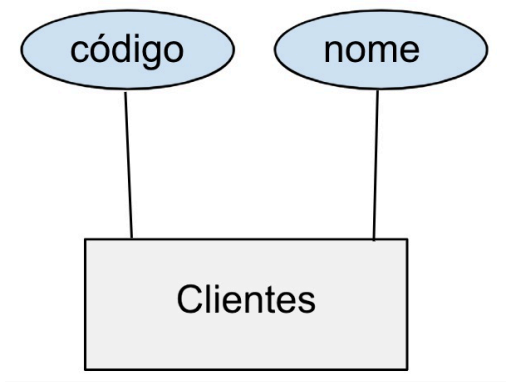

Figura 4: Representação de atributos simples da entidade **"Clientes"**

Os atributos compostos são formados por subatributos, como "Endereço" com os subatributos "Rua", "Cidade" e "Estado". No diagrama ER, os atributos compostos são representados por uma elipse contendo os subatributos.

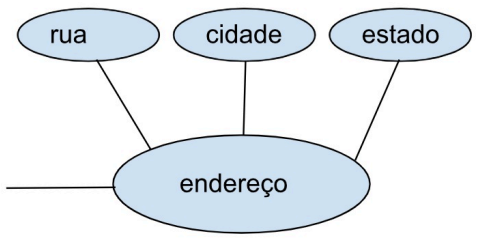

Figura 5: Representação de subatributos de **"endereço"**

Atributos multivalorados são aqueles que podem ter múltiplos valores para uma única instância de uma entidade. Por exemplo, um atributo multivalorado "Telefone" pode ter vários números de telefone associados a um único cliente. No diagrama ER, esses atributos são representados por elipses com linhas duplas.

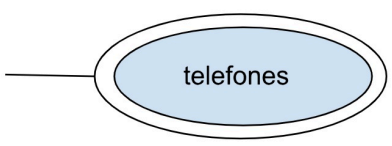

Figura 6: Representação do atributo multivalorado **"telefones"**

Atributos derivados são aqueles cujos valores são calculados a partir de outros atributos. Eles não são armazenados fisicamente no banco de dados, mas podem ser calculados quando necessário. Por exemplo, um atributo derivado "Idade" pode ser calculado a partir do atributo "Data de Nascimento". No diagrama ER, os atributos derivados são representados por elipses com linhas tracejadas.

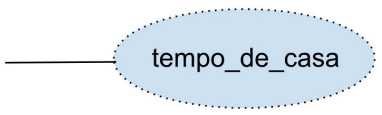

Figura 7: Representação do atributo derivado **"tempo_de_casa"**

Os atributos chaves desempenham um papel essencial na identificação exclusiva das entidades em um conjunto de entidades. A **chave primária** é um atributo (ou conjunto de atributos) que é utilizado para distinguir de forma única cada instância da entidade. Ela é destacada no diagrama ER, geralmente sublinhada, para indicar sua importância como chave primária. Através da chave primária, é possível garantir a unicidade e a identificação precisa de cada entidade no modelo de dados.

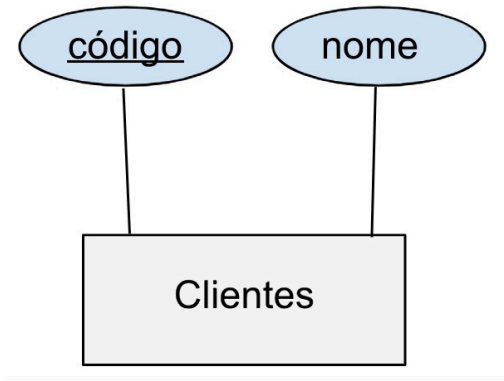

Figura 8: Representação da indicação da chave primária **"código"**

Relacionamentos

No modelo ER, os relacionamentos representam as associações e interações entre entidades. Descrevem como as entidades estão conectadas ou se relacionam umas com as outras. Os relacionamentos são uma parte essencial do projeto de um banco de dados, pois ajudam a modelar a forma como os dados estão relacionados no domínio do problema.

No diagrama ER, os relacionamentos são representados por meio de losangos, que são conectados às entidades envolvidas no relacionamento. Esses losangos são colocados ao longo das linhas que conectam as entidades e servem como símbolos visuais para indicar a existência de um relacionamento. Os relacionamentos são representados por linhas

que conectam as entidades envolvidas.

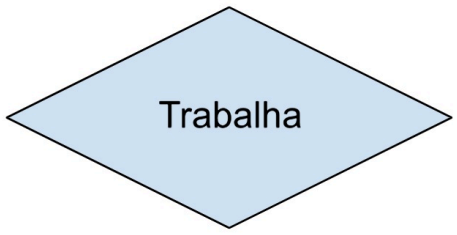

Figura 9: Representação do símbolo de relacionamento

Os atributos de um relacionamento são representados por meio de elipses ligadas ao losango que representa o relacionamento no diagrama ER, e descrevem informações adicionais e exclusivas associadas a esse relacionamento em particular.

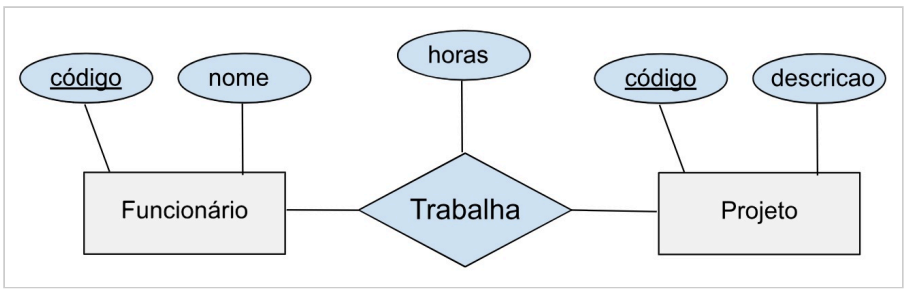

Figura 10: Representação dos atributos de um relacionamento

No modelo ER, o grau do relacionamento refere-se ao número de entidades envolvidas em um relacionamento. O grau pode ser classificado em três categorias principais: relacionamento de grau **binário**, **ternário** e **n-ário**.

Um relacionamento de grau binário envolve duas entidades, sendo o tipo mais comum de relacionamento no modelo ER. Por exemplo, num sistema de gerenciamento de biblioteca, pode haver um relacionamento binário entre as entidades "Livro" e "Autor", representando a associação de um livro com seu autor.

Um relacionamento de grau ternário envolve três entidades. Por exemplo, num sistema de reserva de passagens aéreas, pode haver um relacionamento ternário entre as entidades "Passageiro", "Voo" e "Assento", representando a associação de um passageiro a um voo específico ocupando um assento específico.

É importante observar que um relacionamento com grau N > 2 só é justificável se não puder ser decomposto em relacionamentos com graus menores e ainda manter a semântica desejada.

Autorrelacionamento

Autorrelacionamento no modelo ER, ocorre quando uma entidade se relaciona com instâncias da mesma entidade. Isso é útil para representar interações dentro da mesma categoria, como hierarquias em uma organização. O diagrama ER usa um losango para representar um autorrelacionamento, que pode ser rotulado para indicar papéis específicos, como "supervisor" e "subordinado," tornando o modelo de dados mais claro e preciso.

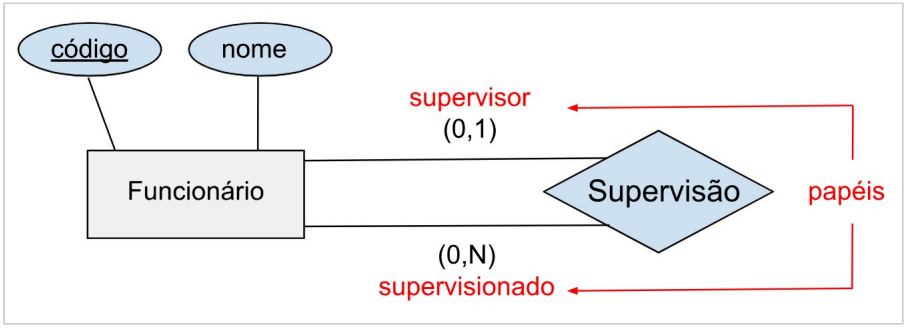

Figura 11: Representação de autorelacionamento

Cardinalidade

Os relacionamentos no modelo de dados podem ser caracterizados pela cardinalidade, a qual expressa quantas ocorrências estão associadas entre as entidades envolvidas. Essa caracterização desempenha um papel fundamental na compreensão das relações entre as entidades e na definição das regras de integridade referencial no banco de dados. A cardinalidade é crucial para determinar como os dados são armazenados e relacionados entre as tabelas, garantindo a coesão e a confiabilidade dos dados. Existem três tipos principais de cardinalidade:

- **Cardinalidade 1:1 (um para um):** uma instância de uma entidade está vinculada a, no máximo, uma instância da outra entidade, e vice-versa.

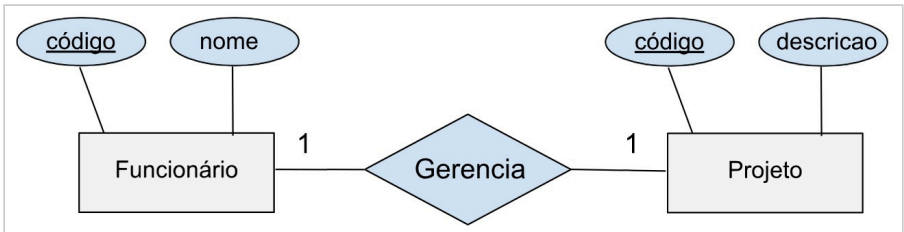

Figura 12: Representação de cardinalidade 1:1

- **Cardinalidade 1:N (um para muitos):** uma instância de uma entidade está relacionada a uma ou mais instâncias da outra entidade, enquanto uma instância da segunda entidade está associada a, no máximo, uma instância da primeira entidade.

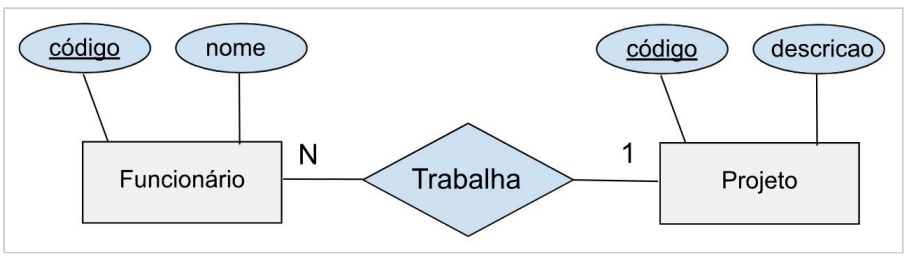

Figura 13: Representação de cardinalidade 1:N

- **Cardinalidade N:N (muitos para muitos):** uma instância de uma entidade pode se conectar com várias instâncias da outra entidade, e vice-versa.

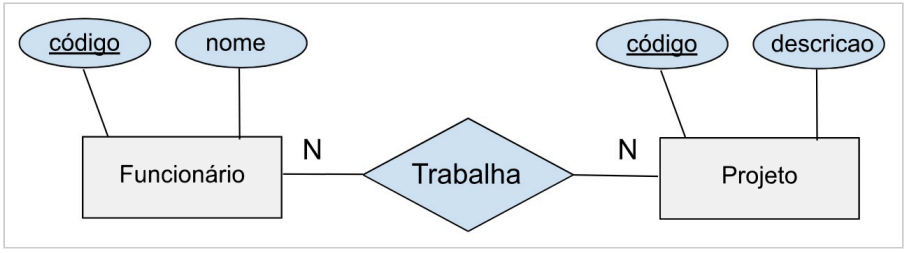

Figura 14: Representação de cardinalidade N:N

As cardinalidades em um modelo de dados podem ser descritas em termos de mínimo e máximo, indicando o número mínimo e máximo de ocorrências que uma entidade pode ter em relação a outra. O mínimo pode ser **"zero**" ou **"um"**, indicando se a presença da entidade é opcional ou obrigatória no relacionamento. O máximo pode ser **"um"** ou **"N"**, representando quantas vezes a entidade pode se relacionar com a outra. Essas especificações permitem definir com precisão a natureza dos relacionamentos e assegurar a integridade e consistência dos dados no banco de dados.

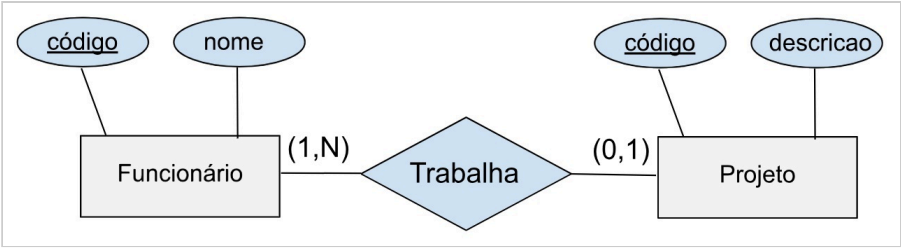

Figura 15: Representação de cardinalidades descritas em termos de ocorrências mínimas e máximas

Entidade Fraca

Uma entidade fraca no modelo ER, é uma entidade que depende de outra entidade para existir. Ela não possui uma chave primária própria e requer a participação de outra entidade, chamada de entidade proprietária, para ser identificada.

A entidade fraca não pode existir independentemente da entidade proprietária, pois sua existência está condicionada à existência da entidade proprietária. Isso significa que a entidade fraca possui uma restrição de participação, também conhecida como restrição de existência total. Essa restrição estabelece que toda instância da entidade fraca deve estar associada a uma instância da entidade proprietária.

No diagrama ER, uma entidade fraca é representada por um retângulo duplo, enquanto a entidade proprietária é representada por um retângulo simples. Além disso, a entidade fraca possui atributos próprios, que ajudam a descrever suas características, mas sua identificação depende da combinação dos atributos chave primária da entidade proprietária e dos atributos próprios.

Modelo ER Estendido

O Modelo Entidade-Relacionamento Estendido (ER Estendido) é uma extensão do modelo ER clássico, usado para modelar estruturas de dados em bancos de dados de forma mais abrangente e complexa. Ele introduz recursos adicionais, como herança, generalização/especialização e agregação, para lidar com casos de uso mais sofisticados.

- **Herança:** A herança permite representar relações de especialização/generalização entre entidades. Isso significa que uma entidade mais genérica pode ser subdividida em subentidades mais específicas, sendo representada no diagrama ER por um triângulo isósceles. Um exemplo prático envolve a modelagem de clientes, onde temos a entidade genérica "Cliente" e suas subentidades "Pessoa Física" e "Pessoa Jurídica".
- **Entidade Genérica:** A entidade mais abstrata que pode ser subdividida em subentidades mais específicas. Por exemplo, considere a entidade genérica "Cliente".
- **Subentidades:** Representam as entidades especializadas dentro da herança. No exemplo, você pode ter subentidades como "Pessoa Física" e "Pessoa Jurídica".
- **Triângulo de Especialização/Generalização:** No diagrama ER as subentidades são conectadas à entidade genérica usando uma linha que se origina de um triângulo isósceles. O triângulo indica a relação de especialização/generalização.
- **Atributos Específicos:** Se as subentidades tiverem atributos específicos, você pode representá-los dentro das subentidades. Por exemplo, a "Pessoa Física" pode ter atributos como "PDF" e "Genero", enquanto uma "Pessoa Jurídica" pode ter atributos diferentes.
- **Restrições de Totalidade e Parcialidade:** Determinam como as entidades genéricas e especializadas se relacionam: Se for total, significa que cada entidade genérica (Cliente) deve estar associada a pelo menos uma subentidade (Pessoa Física ou Pessoa Jurídica). Se for parcial, a associação é opcional, permitindo que um Cliente seja apenas

uma Pessoa Física ou uma Pessoa Jurídica, mas não é necessário que seja ambas. No diagrama ER devemos adicionar "T" (total) ou "P" (parcial) próximo ao triângulo para indicar se a associação entre a entidade genérica e as subentidades é obrigatória (total) ou opcional (parcial).

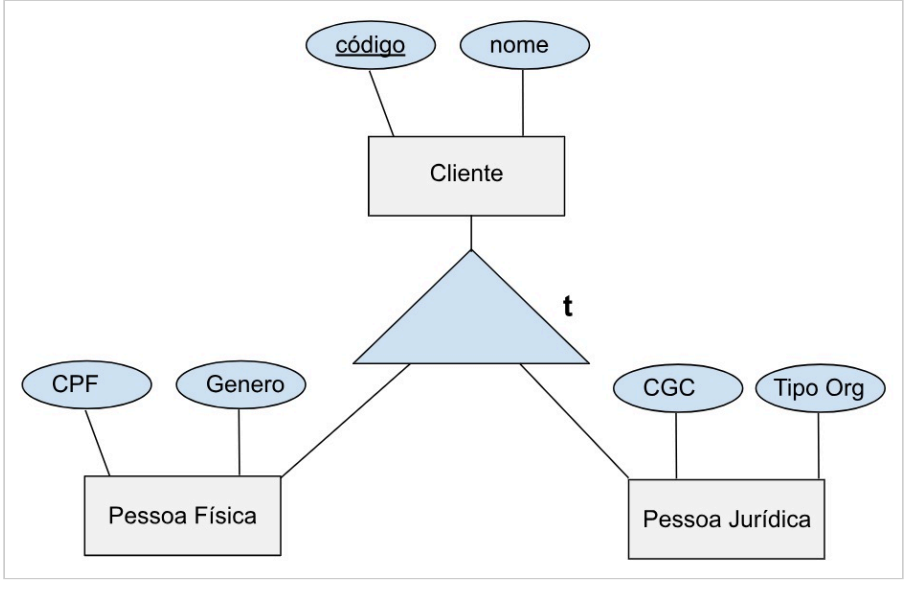

Figura 16: Representação de herança, entidades genéricas, subentidades, generalização e restrição de totalidade

- **Agregação:** A agregação é um recurso fundamental no Modelo Entidade-Relacionamento (ER) que permite combinar entidades relacionadas em uma única entidade agregada de nível superior, representada por um retângulo no diagrama ER. Essa técnica é particularmente útil quando desejamos agrupar informações relacionadas para facilitar a modelagem e a compreensão do sistema. Por exemplo, a entidade **"Consulta Médica"** representada na Figura 17, é uma entidade agregada que reúne informações relacionadas a consultas médicas, incluindo detalhes do médico e do paciente. Além disso, a entidade **"Receita"** está associada à entidade **"Consulta Médica"**para representar as receitas médicas emitidas durante uma consulta. A agregação permite uma organização eficiente de informações complexas relacionadas a consultas médicas em um único bloco, tornando a modelagem mais clara e compreensível.

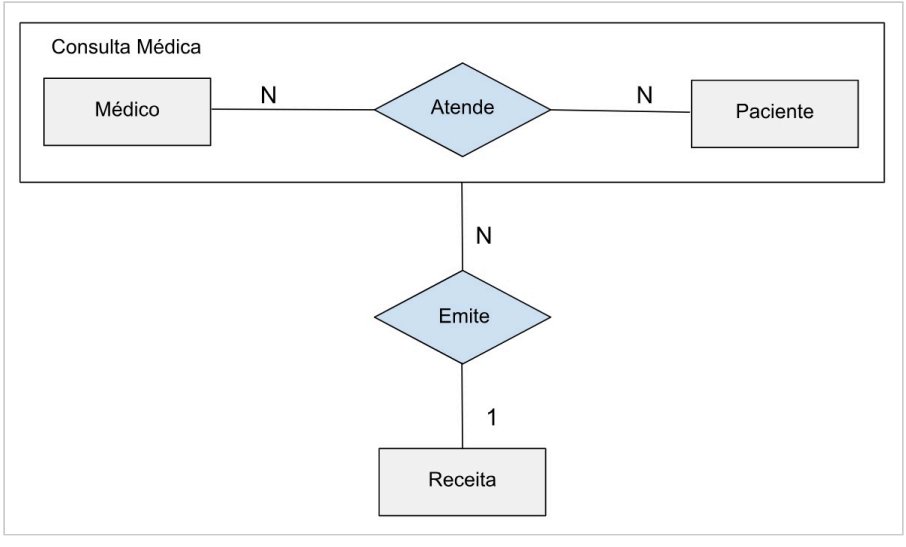

Figura 17: Representação agregação

Essas extensões fornecem uma representação mais abrangente e precisa de relacionamentos complexos e hierarquias em bancos de dados.

Modelo Lógico

O modelo lógico de um banco de dados descreve a organização, o armazenamento e o acesso aos dados em um SGBD específico, adaptando-se às características e funcionalidades desse sistema. Um dos modelos lógicos mais preeminentes é o modelo relacional, introduzido por *Ted Codd* em 1970, que ganhou ampla adoção devido à sua simplicidade e fundamentação matemática.

Em um banco de dados relacional, os dados são organizados em coleções de tabelas, cada uma delas com um nome exclusivo. Cada linha em uma tabela representa um relacionamento entre um conjunto de valores, correspondendo a uma entidade ou relação do mundo real. O conceito de tabela estabelece uma ligação direta com a relação matemática subjacente.

No modelo relacional, uma linha é denominada **tupla** e representa uma sequência de valores. Um relacionamento entre **n** valores é representado por uma **n-upla** de valores, correspondendo a uma linha na tabela. Cada coluna da tabela é referida como atributo. A ordem das tuplas em uma relação é irrelevante, pois uma relação é um conjunto de tuplas.

Cada atributo de uma relação possui um domínio, que é o conjunto de valores permitidos para esse atributo. No modelo relacional, os domínios são exigidos como sendo atômicos, ou seja, unidades indivisíveis. Por exemplo, se um atributo de uma tabela é "número de telefone", o domínio desse atributo é considerado atômico, tratando cada número de telefone como uma unidade indivisível.

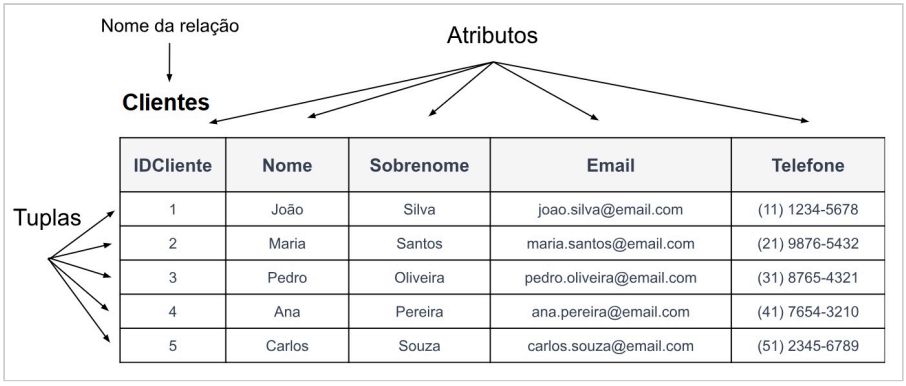

Figura 18: Os atributos e tuplas de uma relação **Cliente** - Adaptado de (ELMASRI; NAVATHE, 2011)

Exemplo: Tabela Pedidos

Tabela 1: Exemplo de Tabela de Pedidos

ID Pedido	Data do Pedido	Cliente	Total do Pedido
1	12/09/2023	1	R$ 250,00
2	11/09/2023	3	R$ 120,00
3	10/09/2023	2	R$ 75,00
4	09/09/2023	5	R$ 350,00
5	08/09/2023	1	R$ 60,00

Nos exemplos, a chave primária da tabela **"Clientes"** é o atributo **"IDCliente"**, enquanto a chave estrangeira na tabela **"Pedidos"** é o atributo **"Cliente"**, que se relaciona com a chave primária da tabela **"Clientes"**. Isso permite estabelecer relacionamentos entre clientes e pedidos. As restrições de integridade garantem que os dados sejam consistentes e válidos em todo o banco de dados relacional.

Você Sabia?

O termo "relacional" no modelo relacional não se refere diretamente às relações ou relacionamentos entre tabelas, como muitos poderiam pensar inicialmente. Em vez disso, ele tem suas raízes na teoria dos conjuntos e na lógica matemática.

Chaves e Restrições de Integridade

- **Chave Primária (PK):** Uma chave primária é um conjunto de atributos que identifica de forma única cada tupla em uma relação. Ela garante que não haja duplicação de registros na tabela. Por exemplo, em uma tabela de **"Clientes"**, o número de identificação do cliente (**IDCliente**) pode ser definido como a chave primária.

- **Chave Candidata:** Uma chave candidata é um conjunto de atributos que também poderia ser escolhido como chave primária, ou seja, ela é única e identifica cada tupla de forma exclusiva. Uma tabela pode ter várias chaves candidatas. No exemplo anterior, o número de telefone de um cliente poderia ser uma chave candidata, desde que seja único para cada cliente.

- **Chave Estrangeira (FK):** Uma **chave estrangeira** é um atributo ou conjunto de atributos em uma tabela que se refere à chave primária de outra tabela. Isso estabelece uma relação entre as tabelas. Por exemplo, em uma tabela de **"Pedidos"**, o atributo **"Cliente"** pode ser uma chave estrangeira que faz referência à chave primária da tabela **"Clientes"**.

Restrições de Integridade

As restrições de integridade desempenham um papel crucial no modelo relacional para garantir a qualidade e a consistência dos dados. Algumas restrições comuns incluem:

- **Restrição de Integridade de Entidade:** Garante que cada tupla em uma tabela seja única, com base na chave primária.

- **Restrição de Integridade Referencial:** Garante que os valores em uma chave estrangeira correspondam a valores na chave primária da tabela relacionada.

- **Restrição de Integridade de Domínio:** Define os valores permitidos para um atributo com base no seu domínio.

- **Restrição de Integridade de *Check*:** Permite que regras personalizadas sejam aplicadas a valores em um atributo.

- **Restrição de Integridade de Valor Padrão:** Define um valor padrão para um atributo caso nenhum valor seja especificado.

- **Restrição de *Not Null* (NN):** Impede que um atributo contenha valores nulos, assegurando que todos os campos obrigatórios sejam preenchidos.

- **Restrição de Unicidade (*Unique*):** Garante que os valores em uma coluna ou conjunto de colunas sejam exclusivos, mas não necessariamente uma chave primária.

Essas restrições de integridade ajudam a manter a qualidade dos dados e a consistência das relações entre tabelas em um banco de dados relacional.

Mapeamento do Modelo Entidade-Relacionamento (ER) para o modelo relacional

O processo de mapeamento do Modelo Entidade-Relacionamento (ER) para o modelo relacional é uma das etapas mais importantes no processo de modelagem de banco de dados. É fundamental para traduzir a estrutura conceitual do banco de dados, representada pelo ER, em uma implementação prática no modelo relacional.

O modelo ER é uma representação gráfica das entidades, relacionamentos e atributos de um sistema, enquanto o modelo relacional é uma representação tabular dos dados. O mapeamento envolve a transformação das entidades, relacionamentos e atributos do modelo ER em tabelas, colunas e chaves primárias e estrangeiras no modelo relacional.

Por exemplo, uma entidade no modelo ER pode ser mapeada para uma tabela no modelo relacional, enquanto um relacionamento pode ser mapeado para uma chave estrangeira em uma tabela relacionada. O mapeamento do modelo ER para o modelo relacional é geralmente automatizado ou semiautomatizado por ferramentas de projeto de banco de dados, o que ajuda a garantir a precisão e consistência do modelo relacional. O modelo relacional resultante é então implementado em um SGBD para armazenar e gerenciar os dados do sistema (ELMASRI; NAVATHE, 2011).

A seguir, delineamos as oito etapas centrais deste processo. A Figura 19 fornece um resumo visual das regras de mapeamento que são fundamentais para traduzir efetivamente o Modelo ER em um modelo relacional prático

- **Etapa 1:** Para cada entidade E no modelo ER, é criada uma tabela T1 no modelo relacional. A tabela T1 inclui todos os atributos simples de E, e um dos atributos chaves de E é escolhido como a chave primária de T1.
- **Etapa 2:** Para cada entidade fraca EF com entidade proprietária E no modelo ER, é criada uma tabela T1 no modelo relacional. A tabela T1 inclui todos os atributos simples de EF, e a chave primária de T1 é composta pela chave parcial de EF e a chave primária de E.
- **Etapa 3:** Para cada relacionamento regular com cardinalidade 1:1 entre as entidades E1 e E2, que geraram as tabelas T1 e T2 respectivamente, escolhe-se a chave primária de uma das relações e a insere-se como chave estrangeira na outra relação. Se houver participação total em um lado do relacionamento e parcial no outro, a chave do lado com participação parcial é inserida como chave estrangeira no lado com participação total.
- **Etapa 4:** Para cada relacionamento regular com cardinalidade 1:N entre as entidades E1 e E2, que geraram as tabelas T1 e T2 respectivamente, insere-se a chave primária de T1 como chave estrangeira em T2.
- **Etapa 5:** Para cada relacionamento regular com cardinalidade N:N entre as entidades E1 e E2, cria-se uma nova tabela T1 que contém todos os atributos do relacionamento, além dos atributos chave de E1 e E2. A chave primária de T1 é composta pelos atributos chave de E1 e E2.
- **Etapa 6:** Para cada relacionamento n-ário (n > 2), cria-se uma tabela T1 que contém todos os atributos do relacionamento. A chave primária de T1 é composta pelos atributos chaves das entidades participantes do relacionamento.
- **Etapa 7:** Para atributos multivalorados, há duas abordagens possíveis. Se houver uma estimativa do número de ocorrências, adicionam-se colunas extras na tabela existente. Caso o número de ocorrências seja indefinido, cria-se uma nova tabela separada com uma chave estrangeira que referencia a chave primária da entidade correspondente.
- **Etapa 8:** Para mapeamento de especializações, existem três alternativas. Pode-se criar uma tabela única que inclui a entidade genérica e suas especializações. Também é possível criar tabelas separadas para a entidade genérica e cada especialização. Por fim, é possível criar tabelas apenas para as entidades especializadas.

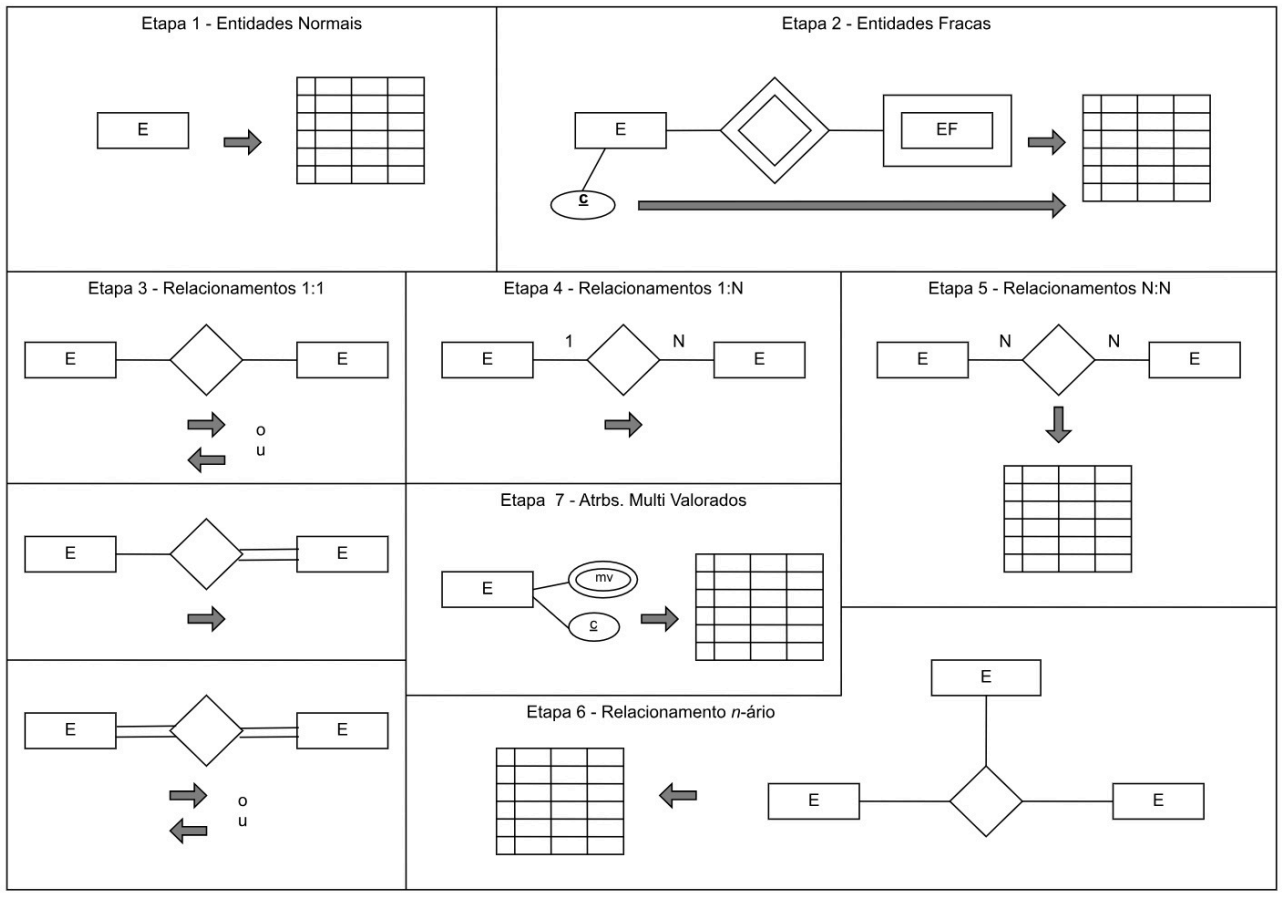

Figura 19: Resumo de regras de mapeamento - Adaptado de (SOUZA, 2019)

Exemplo de Mapeamento

A seguir é apresentado um exemplo de mapeamento de um modelo ER para relacional, considerando o sistema de gerenciamento de projetos.

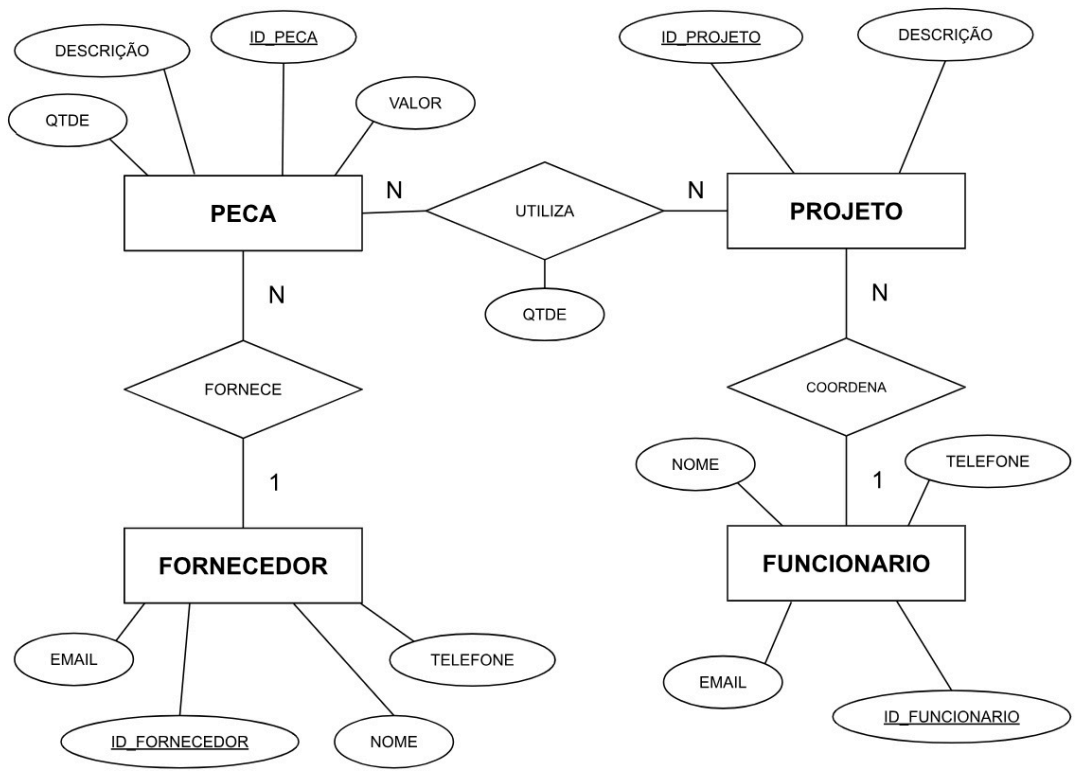

Figura 20: Exemplo de mapeamento de um modelo ER para relacional

- **Etapa 1:** Entidades Fortes

  PECA (ID_PECA (PK), DESCRICAO, QTDE, VALOR)

  FORNECEDOR (ID_FORNECEDOR, NOME, EMAIL, TELEFONE)

  FUNCIONÁRIO (ID_FUNCIONARIO (PK), NOME, TELEFONE, EMAIL)

  PROJETO (ID_PROJETO (PK), DESCRICAO)

- **Etapa 2:** Não há entidades fracas.

- **Etapa 3:** Não há relacionamentos 1:1.

- **Etapa 4:** Relacionamentos 1:N. Inserir a chave primária como chave estrangeira.

  FORNECEDOR e **PECA** (1:N) - Inserir o **ID_FORNECEDOR** (FK) na tabela **PECA**.

  FUNCIONARIO e **PROJETO** (1:N) - Inserir o **ID_FUNCIONARIO** (FK) na tabela **PROJETO**.

- **Etapa 5:** Relacionamentos N:N. Criar uma nova tabela que contém todos os atributos do relacionamento, além dos atributos chaves.

  PECA e **PROJETO** (N:N) - Criar tabela **PECA_PROJETO (ID_PECA (FK)**, **ID_PROJETO (FK)**, **QTDE**, **ID_PECA_PROJETO (PK))**

- **Etapa 6:** Não há relacionamentos n-ários.

- **Etapa 7:** Não há nenhum atributo multivalorado.

- **Etapa 8:** Não há especializações para mapeamento.

Linguagens de Consulta Relacional

As linguagens de consulta relacional desempenham um papel fundamental na interação com bancos de dados relacionais, permitindo que os usuários recuperem, atualizem e gerenciem os dados armazenados nas tabelas do banco de dados. Elas podem ser divididas em duas abordagens principais: linguagens de consulta procedural e linguagens de consulta não procedural. A abordagem procedural envolve a especificação detalhada dos passos necessários para obter um resultado desejado, enquanto a abordagem não procedural concentra-se no resultado final sem descrever os passos intermediários.

A Álgebra Relacional é uma linguagem de consulta procedural que fornece uma base matemática sólida para manipular relações e tabelas em bancos de dados relacionais. Ela é composta por um conjunto de operações que podem ser aplicadas a relações (ou tabelas) para obter os resultados desejados. As principais operações da Álgebra Relacional incluem:

- **Seleção (σ):** Esta operação permite extrair tuplas (linhas) de uma relação que satisfaçam determinadas condições ou critérios especificados. É representada pelo símbolo **"σ"** (sigma) e usada da seguinte forma: **σ<condição>(R)**, onde **"R"** é a relação e **"condição"** é uma expressão lógica que as tuplas selecionadas devem atender.

Selecionar todos os clientes que moram na cidade de Pirapozinho

$$\sigma_{cidade="Pirapozinho"}(Cliente)$$

- **Projeção (π):** A operação de projeção é usada para selecionar um subconjunto de colunas de uma relação. É representada pelo símbolo **"π"** (pi) e usada da seguinte forma: **π<lista de atributos>(R)**, onde **"R"** é a relação e a **"lista de atributos"** é o conjunto de colunas que deve ser mantido no resultado.

Selecionar os atributos nome e cidade de todos os clientes

$$\pi_{nome,cidade}(Cliente)$$

- **União (∪):** A operação de união combina duas relações compatíveis (com o mesmo número e tipo de atributos) e retorna uma nova relação que contém todas as tuplas de ambas as relações, eliminando tuplas duplicadas.

Unir os cadastros de clientes antigos com os de clientes novos

$$Cliente \cup ClienteNovo$$

- **Interseção (∩):** A operação de interseção retorna uma nova relação que contém apenas as tuplas presentes em ambas as relações **R** e **S**.

Verificar cadastros duplicados em clientes antigos e clientes novos

$$Cliente \cap ClienteNovo$$

- **Diferença (−):** A operação de diferença retorna uma nova relação que contém as tuplas presentes em **R**, mas não em **S**.

Verificar quais clientes não realizaram pedido

$$\pi_{IDCliente,Nome}(Cliente) - \pi_{Cliente}(Pedido)$$

- **Produto Cartesiano (×):** O produto cartesiano combina cada tupla de uma relação com todas as tuplas de outra relação, resultando em uma nova relação que contém todas as possíveis combinações de tuplas das duas relações.

Verificar as combinações possíveis entre clientes e pedidos

$$Cliente \times Pedido$$

- **Junção (⋈):** A junção combina tuplas de duas relações com base em uma condição de igualdade especificada, resultando em uma nova relação.

Realizar a junção de clientes e pedidos com base no IDCliente do cliente e Cliente do pedido

$$Cliente \bowtie_{IDCliente=Cliente} (Pedido)$$

Essas operações formam a base para a construção de consultas mais complexas em álgebra relacional, permitindo que os usuários realizem tarefas como filtrar dados, combinar informações de diferentes tabelas e realizar operações matemáticas nos dados. A álgebra relacional é uma ferramenta poderosa para consultas em bancos de dados relacionais, proporcionando flexibilidade e precisão na recuperação de informações.

Modelo Físico (ou de implementação)

O Modelo Físico é a etapa em que um sistema de banco de dados deixa de ser uma ideia abstrata e começa a se tornar uma realidade física. Nesta fase, os conceitos definidos no modelo lógico de dados são traduzidos em estruturas tangíveis que permitem o armazenamento, gerenciamento e recuperação eficaz de dados.

Uma das decisões cruciais nesta fase é a escolha do *software* SGBD que será responsável por gerenciar o armazenamento, a recuperação e a manipulação dos dados. Existem várias opções disponíveis no mercado, e a escolha deve ser feita com base nos requisitos específicos do projeto e nas necessidades do sistema. Cada SGBD tem suas próprias características, adequando-se a diferentes requisitos. Alguns exemplos comuns incluem:

- ***MySQL*:** Amplamente utilizado em aplicações *Web* e projetos de médio porte, destaca-se pela simplicidade e escalabilidade - <https://www.mysql.com/downloads/>.
- ***PostgreSQL*:** Conhecido por sua confiabilidade e capacidade de lidar com volumes de dados significativos, é uma escolha sólida para aplicações complexas - <https://www.postgresql.org/download/>.
- ***Oracle Database*:** Oferece recursos avançados e é comumente escolhido para empresas com demandas de alta complexidade - <https://www.oracle.com/br/downloads/>.
- ***Microsoft SQL Server*:** Amplamente adotado em ambientes corporativos, oferecendo integração com outras ferramentas da *Microsoft* - <https://www.microsoft.com/pt-br/sql-server/>.

Integração com o Modelo Lógico

A integração eficaz entre o modelo físico e o modelo lógico desempenha um papel vital para assegurar que a estrutura física do banco de dados represente de maneira precisa a estrutura conceitual delineada no Modelo Lógico. Embora os comandos SQL específicos sejam discutidos na próxima seção, vale ressaltar que os exemplos apresentados nos passos envolvidos nessa integração são destinados a consolidar os conceitos e a sintaxe apropriada, que serão explorados em maior detalhe posteriormente.

Mapeamento de Entidades para Tabelas

Cada entidade definida no Modelo Lógico deve ser mapeada para uma tabela no banco de dados físico. Isso envolve determinar quais atributos da entidade se tornarão colunas na tabela correspondente. Além disso, é importante especificar os tipos de dados apropriados para cada coluna, garantindo que eles se alinhem com as definições do Modelo Lógico.

Você Sabia?

A transição do modelo conceitual para o modelo lógico é como a tradução entre idiomas. O desafio é capturar a essência do "idioma" original (**modelo conceitual**) e adaptá-lo fielmente ao "idioma" técnico do SGBD (**modelo lógico**). Essa "tradução" é uma combinação de arte e ciência, essencial para um *design* de banco de dados eficaz.

Exemplo: Se o Modelo Lógico inclui uma entidade **"Cliente"** com atributos como **"ID"**, **"Nome"** e **"Email"**, o Modelo de Implementação deve criar uma tabela **"Clientes"** com as colunas correspondentes.

```
CREATE TABLE Clientes (
   ID INT PRIMARY KEY,
   Nome VARCHAR(50),
   Email VARCHAR(100)
);
```

Estabelecimento de Relacionamentos

Se o modelo lógico incluir relacionamentos entre entidades (por exemplo, um cliente que faz vários pedidos), esses relacionamentos devem ser traduzidos em chaves estrangeiras no banco de dados físico. Isso conecta as tabelas e permite que o SGBD mantenha a integridade referencial entre elas.

Exemplo: Se o modelo lógico estabelece que um **"Pedido"** está relacionado a um **"Cliente"**, o Modelo de Implementação criará uma chave estrangeira na tabela **"Pedidos"** que faz referência à tabela **"Clientes"**.

```
CREATE TABLE Pedidos (
   NumeroPedido INT PRIMARY KEY,
   ClienteID INT,
   FOREIGN KEY (ClienteID) REFERENCES Clientes(ID)
);
```

A escolha do SGBD deve estar alinhada com o modelo lógico de dados estabelecido anteriormente. Isso assegura que a estrutura do banco de dados no SGBD reflita as entidades, relacionamentos e restrições definidos no modelo lógico.

Exemplo: Se o modelo lógico inclui entidades como **"Clientes"** e **"Pedidos"**, devemos criar tabelas correspondentes no SGBD para armazenar esses dados.

Tradução de Restrições e Regras de Negócios

As restrições de integridade, validações e regras de negócios definidas no modelo lógico também devem ser traduzidas para o banco de dados físico. Isso garante que o banco de dados aplique essas restrições para manter a consistência e a precisão dos dados.

Exemplo: Se o modelo lógico especifica que o campo **"Email"** na entidade **"Cliente"** deve ser único, o modelo físico deve incluir essa restrição na tabela **"Clientes"**.

```
CREATE TABLE Clientes (
   ID INT PRIMARY KEY,
   Nome VARCHAR(50),
   Email VARCHAR(100) UNIQUE
);
```

Definição de Índices

Com base nas consultas previstas e nos requisitos de desempenho, é necessário decidir quais campos precisam de índices no banco de dados físico. Os índices aceleram a recuperação de dados, mas devem ser aplicados com discernimento para evitar sobrecargas no sistema.

Exemplo: Se consultas frequentes forem realizadas com base no nome do cliente, pode ser apropriado criar um índice no campo **"Nome"** da tabela **"Clientes"**.

```
CREATE INDEX idx_nome ON Clientes(Nome);
```

Otimização e Implementação

No âmbito do modelo físico, a otimização do desempenho assume uma importância crítica. Esse processo engloba uma série de ações, incluindo a configuração de parâmetros no SGBD, o estabelecimento de políticas de *cache* eficazes, a implementação de estratégias de compactação e a alocação adequada de recursos para atender às demandas específicas do sistema.

Por exemplo, ao considerarmos o *MySQL*, é possível otimizar o desempenho de leitura e gravação ajustando o tamanho do *buffer* de *cache* do *InnoDB* [1]. Essa medida pode resultar em melhorias significativas no acesso aos dados, proporcionando uma experiência mais ágil e eficiente para os usuários do sistema.

---

[1] É um "*Storage Engine*", ou seja, um mecanismo de armazenamento distribuído que fornece as funcionalidades padrões de transação flexíveis ao ACID, juntamente com o suporte a chaves estrangeiras.

3. LINGUAGEM SQL

A linguagem SQL é fundamental para a manipulação e administração de dados em sistemas de bancos de dados relacionais. Sendo uma linguagem padronizada de consulta a bancos de dados, ela é empregada em grande parte dos SGBDs disponíveis no mercado. A habilidade de compreender e escrever consultas SQL é indispensável para profissionais que lidam com dados, especialmente aqueles envolvidos em funções de análise, desenvolvimento e administração de bancos de dados.

Origens e Evolução da SQL

A SQL, que surgiu nos primeiros anos da década de 1970 como um projeto iniciado pelo Departamento de Pesquisas da IBM, foi concebida como uma resposta à necessidade de uma linguagem universal para a manipulação de dados em sistemas de bancos de dados relacionais. Seu objetivo primordial era fornecer uma maneira eficaz e padronizada de realizar consultas, inserções, atualizações e exclusões em grandes volumes de dados, sem a necessidade de se preocupar com as complexidades do armazenamento físico.

Com o passar do tempo, a SQL evoluiu para atender às crescentes demandas, e em 1986, o padrão SQL foi formalizado pelo ANSI - *American National Standard Institute*, consolidando-a como a linguagem padrão para sistemas de bancos de dados relacionais.

Estrutura e Componentes Principais da SQL

A SQL é uma linguagem declarativa, o que significa que os usuários a empregam para declarar o que desejam fazer com os dados, em vez de como fazer. Ela abrange várias categorias de comandos que abordam todas as etapas do gerenciamento de dados, e os comandos básicos podem ser divididos em duas categorias principais:

- **DDL (*Data Definition Language*):** A Linguagem de Definição de Dados, inclui comandos como **Create**, **Alter**, **Drop** e **Rename**, que são utilizados para definir a estrutura dos dados, como tabelas, índices e restrições.
- **DML (*Data Manipulation Language*):** A Linguagem de Manipulação de Dados, inclui comandos como **Select**, **Insert**, **Update** e **Delete**, que fazem parte dessa categoria, permitindo aos usuários realizar operações de consulta e manipulação nos dados.

Essas duas categorias desempenham papéis fundamentais no gerenciamento e interação com bancos de dados relacionais, permitindo que os usuários definam a estrutura dos dados e realizem operações para recuperar, adicionar, atualizar ou excluir informações conforme necessário.

A Flexibilidade e o Poder da SQL

Uma das principais vantagens da SQL reside em sua capacidade de operar com flexibilidade em uma ampla gama de ambientes e SGBDs. Desde sistemas de código aberto, como o *MySQL*, até sistemas corporativos complexos, a SQL se adapta para atender às necessidades específicas de cada contexto.

Além disso, a SQL viabiliza a criação de visões, que oferecem diferentes perspectivas dos dados, e também proporciona mecanismos de transações como *Commit* [1] e *Rollback* [2], que garantem a consistência e a integridade dos dados, mesmo durante operações complexas.

SQL na Prática

Apesar de existirem ferramentas visuais disponíveis para a conexão e administração de SGBDs, neste curso nosso enfoque será direcionado para os comandos SQL baseados no *MySQL*, um dos SGBDs mais populares que faz uso extensivo da SQL.

Embora as ferramentas gráficas sejam úteis e amplamente utilizadas para tarefas cotidianas, entender e dominar os comandos SQL é essencial para qualquer profissional de banco de dados.

Ao compreender os comandos SQL, você estará preparado para:

- Realizar tarefas avançadas de administração e manutenção do banco de dados.
- Escrever consultas complexas para extrair informações específicas dos dados.
- Solucionar problemas e otimizar o desempenho do banco de dados.
- Adaptar-se a diferentes SGBDs, uma vez que a maioria deles suporta a linguagem SQL.

Portanto, enquanto as ferramentas visuais podem ser um ponto de partida conveniente, nosso objetivo principal é capacitá-lo(a) com o conhecimento necessário para interagir com bancos de dados de forma eficaz e independente, através dos comandos SQL.

Você Sabia?

A sigla SQL significa "*Structured Query Language*", mas curiosamente, quando foi inicialmente desenvolvida nos laboratórios da IBM nos anos 1970, era chamada de **"SEQUEL"**, que significa "*Structured English Query Language*". O nome teve que ser alterado devido a questões de direitos autorais com outra empresa, mas a linguagem manteve sua essência e tornou-se o padrão de fato para gerenciamento de bancos de dados.

[1] Uma transação efetuada com sucesso, onde todas as suas operações foram bem-sucedidas, alterando o banco de dados de forma permanente e com os dados envolvidos na transação sendo persistidos, ou seja, salvos em disco.

[2] Operação realizada quando ocorre uma falha durante o precesso de gravação de informações num ou mais bancos de dado (por exemplo por uma queda de energia). Tal operação retoranará então o(s) banco(s) afetado(s), para o seu estado antes da falha.

Criação de Tabelas e Carga de Dados

Antes de começar a criar tabelas no *MySQL*, é fundamental compreender como criar um banco de dados no qual essas tabelas serão armazenadas.

Criação de um Banco de Dados

O comando **CREATE DATABASE** permite criar um espaço de armazenamento lógico onde as tabelas, os dados e outros objetos do banco de dados serão organizados e gerenciados.

Sintaxe básica

```
CREATE DATABASE nome_do_banco;
```

Exemplo: Criar um banco de dados chamado **"LojaDeGames"**

```
CREATE DATABASE LojaDeGames;
```

Verificação da Criação do Banco de Dados

Para confirmar se o banco de dados foi criado com êxito, listar todos os bancos de dados existentes utilizando o comando **SHOW DATABASES**. Dessa forma, será exibida uma lista contendo todos os bancos de dados, inclusive o recém criado.

Sintaxe básica

```
SHOW DATABASES;
```

Seleção do Banco de Dados

Para começar a trabalhar com o banco de dados recém-criado, ou já existente, é necessário selecioná-lo utilizando o comando USE, seguido do nome do banco de dados desejado. A seleção de um banco de dados é uma etapa crucial, pois define qual banco de dados será o contexto ativo para as operações subsequentes, como criação de tabelas, inserção de dados e consultas.

Sintaxe básica

```
USE LojaDeGames;
```

Após a execução bem-sucedida do comando **USE**, o banco de dados **"LojaDeGames"** se tornará o banco de dados ativo e qualquer comando SQL subsequente que você execute será aplicado a esse banco de dados.

Se um banco de dados não estiver selecionado com o comando **USE**, você ainda pode acessá-lo por meio de qualificação, referindo-se a ele usando a sintaxe completa que inclui o nome do banco de dados antes do nome da tabela.

```
meu_banco.minha_tabela;
```

Exclusão de um Banco de Dados

O comando **DROP DATABASE** realiza a exclusão permanente de um banco de dados e todos os objetos associados a ele, incluindo tabelas, índices, visões, procedimentos armazenados, entre outros. Essa operação é irreversível e, portanto, deve ser executada com extrema cautela.

Sintaxe básica

```
DROP DATABASE nome_do_banco;
```

Exemplo: Excluir o banco de dados **"LojaDeGames"**

```
DROP DATABASE LojaDeGames;
```

Criação de Tabelas

A criação de tabelas é uma etapa fundamental na construção de um banco de dados. As tabelas são estruturas que organizam os dados de forma lógica, permitindo o armazenamento, recuperação e manipulação eficiente das informações. O comando **CREATE TABLE** permite definir a estrutura da tabela, especificando os nomes das colunas (atributos), os tipos de dados das colunas e outras características, como restrições de integridade referencial.

Sintaxe básica

```
CREATE TABLE nome_da_tabela (
   nome_coluna1 tipo_de_dado1 [restrições],
   nome_coluna2 tipo_de_dado2 [restrições],
   …
   [OUTRAS OPÇÕES]
);
```

- **nome_da_tabela:** O nome da tabela que você deseja criar.
- **nome_da_coluna:** O nome da coluna na tabela.
- **tipo_de_dado:** O tipo de dados que a coluna irá armazenar (por exemplo, **INT**, **VARCHAR**, **DATE**, etc.).
- **[restrições]**: É opcional e pode incluir várias restrições, como **NOT NULL**, **PRIMARY KEY**, **UNIQUE**, **DEFAULT**, **AUTO_INCREMENT**, entre outras.
- **[OUTRAS OPÇÕES]:** Isso pode incluir opções adicionais, como restrições de chave estrangeira e índices.

A Tabela 2 a seguir apresenta os tipos de dados comuns no *MySQL*

Tabela 2: Tipos *MySQL* comuns

Tipo de Dado	Descrição
INT	Número inteiro
VARCHAR(n)	Texto com comprimento variável (até n caracteres)

Tipo de Dado	Descrição
CHAR(n)	Texto com comprimento fixo (exatamente n caracteres)
DATE	Data no formato "AAAA-MM-DD"
TIME	Hora no formato "HH:MM:SS"
DATETIME	Data e hora no formato "AAAA-MM-DD HH:MM:SS"
DECIMAL(p, s)	Número decimal com precisão p e escala s
BOOLEAN	Valor verdadeiro **(TRUE)** ou falso **(FALSE)**

Dica:

Você pode conhecer todos os tipos de dados suportados pelo *MySQL*, acessando a documentação oficial do SGBD, disponível em: <https://dev.mysql.com/doc/refman/8.0/en/data-types.html>.

A Tabela 3 a seguir, descreve as principais restrições que podem ser aplicadas às colunas em uma tabela, e desempenham um papel fundamental na definição das regras que governam o comportamento das colunas e garantem a consistência e a integridade dos dados em um banco de dados.

Tabela 3: Restrições sobre colunas em tabelas

Restrição	Descrição
NOT NULL	Impede que uma coluna aceite valores nulos, ou seja, todos os registros devem ter um valor válido.
PRIMARY KEY	Define uma coluna como a chave primária da tabela, garantindo que cada valor seja exclusivo.
UNIQUE	Garante que todos os valores em uma coluna sejam únicos, mas permite valores nulos.
DEFAULT valor	Define um valor padrão para uma coluna quando nenhum valor é especificado durante a inserção.
AUTO_INCREMENT	Usado geralmente com chaves primárias numéricas para gerar automaticamente valores sequenciais.
FOREIGN KEY	Cria uma chave estrangeira que estabelece uma relação entre as colunas de duas tabelas.

Restrição	Descrição
CHECK (condição)	Permite definir uma condição que os valores em uma coluna devem atender durante a inserção ou atualização.
INDEX	Cria um índice na coluna para melhorar o desempenho em consultas, mas não impõe restrições de integridade.

Você Sabia?

Enquanto muitos consideram a SQL uma linguagem de programação, ela é, na verdade, uma linguagem de consulta ou manipulação de dados. Ao contrário das linguagens de programação tradicionais que são projetadas para construir aplicações completas, a SQL é especializada em consultar, atualizar e gerenciar dados em bancos de dados. Esta especialização tornou-a a linguagem padrão para bancos de dados relacionais por mais de quatro décadas!

A maioria das restrições que envolvem integridade de dados, como **PRIMARY KEY**, **UNIQUE**, **CHECK** e **FOREIGN KEY**, normalmente são aplicadas usando a cláusula **CONSTRAINT** para fornecer nomes específicos às restrições e personalizar suas condições. Restrições como **NOTNULL**, valor **DEFAULT** e **INDEX** podem ser aplicadas diretamente às colunas sem a necessidade de **CONSTRAINT**.

Exemplo: Criação de uma tabela para armazenar informações sobre **"Jogos"**

```
CREATE TABLE Jogos (
   id INT AUTO_INCREMENT PRIMARY KEY,
   nome VARCHAR(255) NOT NULL,
   plataforma VARCHAR(50),
   data_lancamento DATE,
   descricao TEXT
);
```

Exemplo: Criação de tabela para armazenar informações sobre **"Clientes"**

```
CREATE TABLE Clientes (
   id INT AUTO_INCREMENT PRIMARY KEY,
   nome VARCHAR(100) NOT NULL,
   email VARCHAR(255) UNIQUE NOT NULL,
   senha VARCHAR(100) NOT NULL,
   data_nascimento DATE,
   endereco VARCHAR(255),
   cidade VARCHAR(100),
   estado CHAR(2),
   cep VARCHAR(10),
   CONSTRAINT check_cidadex1 CHECK (YEAR(data_nascimento) = 1972)
);
```

Exemplo: Criação da tabela para registrar informações de **"Pedidos"**

```
CREATE TABLE Pedidos (
   id INT AUTO_INCREMENT PRIMARY KEY,
   cliente_id INT NOT NULL,
   data_pedido DATE NOT NULL,
   total DECIMAL(10, 2) NOT NULL,
   status VARCHAR(50) DEFAULT 'Em Processamento',
   CONSTRAINT FK_pedidos FOREIGN KEY (cliente_id) REFERENCES Clientes(id)
);
```

Modificar a Estrutura de uma Tabela

O comando **ALTER TABLE** permite modificar a estrutura de uma tabela existente, permitindo que você adicione, modifique ou remova colunas, índices e restrições.

Sintaxe Geral

```
ALTER TABLE nome_da_tabela;
```

A partir dessa instrução inicial, você pode adicionar várias cláusulas para realizar a operação específica desejada.

Exemplo: Adicionar uma Nova Coluna

```
ALTER TABLE nome_da_tabela
ADD COLUMN nome_da_coluna tipo_de_dado;

ALTER TABLE Clientes
ADD COLUMN telefone VARCHAR(15);
```

Exemplo: Modificar uma Coluna Existente

```
ALTER TABLE nome_da_tabela
MODIFY COLUMN nome_da_coluna novo_tipo_de_dado;

ALTER TABLE Pedidos
MODIFY COLUMN quantidade INT;
```

Você Sabia?

Apesar da predominância global da SQL, sua pronúncia varia. Enquanto alguns dizem "S-Q-L", letra por letra, outros pronunciam como "sequel" - um eco do seu nome original, **SEQUEL**.

Exemplo: Renomear uma Coluna

```
ALTER TABLE nome_da_tabela
CHANGE COLUMN nome_da_coluna novo_nome_da_coluna tipo_de_dado;
```

```
ALTER TABLE Produtos
CHANGE COLUMN descricao descricao_produto TEXT;
```

Exemplo: Remover uma Coluna

```
ALTER TABLE nome_da_tabela
DROP COLUMN nome_da_coluna;

ALTER TABLE Pedidos
DROP COLUMN data_entrega;
```

Inserção/Carga de dados

O comando **INSERT** é uma instrução SQL fundamental usada para adicionar novos registros (ou linhas) a uma tabela em um banco de dados.

Sintaxe básica

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

- **nome_da_tabela:** O nome da tabela na qual você deseja inserir os dados.
- **(coluna1, coluna2, coluna3, ...):** Uma lista opcional de colunas nas quais você deseja inserir valores. Se você não especificar colunas, o **INSERT** assumirá que os valores estão sendo inseridos em todas as colunas na ordem em que elas foram definidas na tabela.
- **VALUES:** A palavra-chave **VALUES** é usada para especificar os valores que serão inseridos nas colunas correspondentes.
- **(valor1, valor2, valor3, ...):** Uma lista de valores correspondentes às colunas especificadas. Cada valor deve corresponder ao tipo de dado e à ordem das colunas.

Exemplo: Carga de dados na tabela Jogos

```
INSERT INTO Jogos (nome, plataforma, data_lancamento, descricao)
VALUES ('FIFA 23', 'PlayStation 5', '2022-09-27', 'O mais recente da série FIFA.');

INSERT INTO Jogos (nome, plataforma, data_lancamento, descricao)
VALUES
   ('Red Dead Redemption 2', 'PlayStation 4', '2018-10-26', 'Um jogo de ação no Velho Oeste.'),
   ('The Elder Scrolls V: Skyrim', 'PC', '2011-11-11', 'Um RPG épico com mundos abertos e dragões.');
```

Considerações importantes a respeito do comando **INSERT**:

- Certifique-se de que os valores inseridos estejam em conformidade com os tipos de dados das colunas.
- É possível inserir múltiplas linhas em uma única instrução **INSERT**, fornecendo várias listas de valores dentro dos parênteses **VALUES**.
- Se uma coluna tiver uma restrição **NOT NULL**, você deve fornecer um valor para essa coluna.
- Ao inserir dados em colunas de tipo de dados *string* (como **VARCHAR** ou **CHAR**), coloque os valores entre aspas simples (ou duplas, dependendo das configurações).

- Se uma coluna tem uma restrição **UNIQUE**, os valores inseridos não podem ser duplicados naquela coluna.
- A ordem dos valores na cláusula **VALUES** deve corresponder à ordem das colunas listadas (ou à ordem em que foram definidas na tabela).
- Use instruções **INSERT** em conjunto com outras cláusulas SQL, como **SELECT**, para copiar dados de uma tabela para outra ou para realizar inserções com base em consultas.

Alteração de Dados de Tabela

O comando **UPDATE** é utilizado para modificar os dados existentes em uma tabela. O comando permite atualizar os valores em uma ou mais colunas de registros específicos que atendam a uma determinada condição.

Sintaxe básica

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

- **nome_da_tabela:** O nome da tabela na qual você deseja realizar as atualizações.
- **SET:** Indica as colunas que você deseja atualizar e seus novos valores.
- **coluna1 = novo_valor1, coluna2 = novo_valor2, ...:** Especifica as colunas que serão atualizadas e os novos valores para essas colunas.
- **WHERE condição:** Define a condição que determina quais registros serão atualizados. Se você não fornecer uma condição, todos os registros da tabela serão atualizados.

Exemplo: Atualizar um valor em uma Coluna

```
UPDATE Clientes
SET cidade = “Pirapozinho”
WHERE id=123;
```

Exemplo: Atualizar múltiplas colunas em um Registro

```
UPDATE Clientes
SET nome = 'João da Silva', email = 'joao_novo@email.com'
WHERE id=456;
```

Exemplo: Atualizar registros com base em uma Condição

```
UPDATE Clientes
SET estado = “SP”
WHERE cidade = “Pirapozinho;
```

Esteja extremamente cauteloso ao usar um **UPDATE** sem uma cláusula **WHERE**, pois isso pode afetar todos os registros da tabela.

Exclusão de Dados de Tabela

O comando **DELETE** é utilizado para remover registros de uma tabela. Permite excluir registros com base em uma determinada condição.

Sintaxe básica

```
DELETE FROM nome_da_tabela
WHERE condição;
```

- **nome_da_tabela:** O nome da tabela da qual você deseja excluir registros.
- **WHERE condição:** A condição que determina quais registros serão excluídos. Se você não fornecer uma condição, todos os registros da tabela serão excluídos.

Exemplo: Excluir um Único Registro com Base na Chave Primária

```
DELETE FROM Clientes
   WHERE id = 123;
```

Exemplo: Excluir Registros com Base em uma Condição

```
DELETE FROM Clientes
   WHERE cidade = “Pirapozinho;
```

Extração de Dados e Relatórios

A instrução **SQL SELECT** é fundamental para extrair dados de tabelas específicas em um banco de dados. Com o **SELECT**, é possível escolher quais colunas você deseja extrair e aplicar critérios de seleção para buscar as linhas que atendem a esses critérios. Além disso, você pode ordenar ou agrupar os resultados de acordo com suas necessidades.

Sintaxe SQL da instrução SELECT:

```
SELECT <colunas>
FROM <tabelas>
ORDER BY <coluna1> [ASC | DESC], <coluna2> [ASC |
DESC], ...;
```

```
SELECT [ ALL | DISTINCT ] <lista de atributos>
FROM <lista de Tabelas>
[ WHERE <condição> ]
[ GROUP BY <Atributo>, ...
[ HAVING <condição> ] ]
ORDER BY <Lista de atributos> [ ASC | DESC ], ...]
```

Exemplo: Listar todos os atributos e tuplas da tabela Jogos

```
SELECT * FROM
Jogos
```

Exemplo: Listar os atributos **nome** e **ano_lancamento** da tabela **Jogos**

```
SELECT nome , ano_lancamento FROM
Jogos
```

Exemplo: Listar os atributos **nome** e **ano_lancamento** da tabela **Jogos**, renomeando o atributo **ano_lancamento** para **"ano"**

```
SELECT nome , ano_lancamento as ano FROM Jogos
```

Cláusula WHERE

A cláusula **WHERE** é usada para aplicar filtros aos resultados da instrução **SELECT** , permitindo que você especifique quais registros das tabelas listadas na cláusula **FROM** são afetados pela consulta. Sem a cláusula **WHERE** , a consulta retornará todas as linhas da tabela, mas ao utilizá-la, você pode destacar operadores lógicos (como **AND** , **OR** e **NOT**) e operadores de comparação (como **"="** , **"<"** , **">"** , **"<="** , **">="** , **"<>"**) para restringir os resultados com base em critérios específicos.

Exemplo: Listar jogos lançados após 2020

```
SELECT nome, ano_lancamento
FROM Jogos
WHERE ano_lancamento > 2020;
```

Exemplo: Selecionar todos os jogos cujo ano de lançamento está entre 2020 e 2023.

```
SELECT nome, ano_lancamento
FROM Jogos
WHERE ano_lancamento >= 2020 AND ano_lancamento <= 2023;
```

Operador BETWEEN

Para selecionar valores de um atributo que estejam dentro de um intervalo especificado. É uma maneira conveniente de simplificar consultas que envolvem intervalos.

Exemplo: Selecionar todos os produtos cujo ano de lançamento está entre 2020 e 2023.

```
SELECT nome, ano_lancamento
FROM Jogos
WHERE ano_lancamento BETWEEN 2020 AND 2023;
```

Operador IS NULL / IS NOT NULL

Para verificar se um atributo contém valores nulos (ausência de valor - **IS NULL**) ou não nulos (um valor está presente - **IS NOT NULL**), respectivamente.

Exemplo: Selecionar todos os registros da tabela de jogos cujo atributo ano de lançamento esteja nulo

```
SELECT nome, ano_lancamento
FROM Jogos
WHERE ano_lancamento IS NULL;
```

Exemplo: Selecionar todos os registros da tabela de jogos cujo atributo ano de lançamento não seja nulo

```
SELECT nome, ano_lancamento
FROM Jogos
WHERE ano_lancamento IS NOT NULL;
```

Você Sabia?

Embora a SQL seja famosa por gerenciar bancos de dados relacionais, variantes da SQL foram desenvolvidas para bancos de dados *NoSQL*, mostrando a adaptabilidade e influência duradoura da linguagem em diferentes paradigmas de banco de dados.

Cláusula GROUP BY

A cláusula **GROUP BY** é usada para combinar (agrupar dados) registros com valores idênticos na lista de campos especificados em um único registro. Você pode criar valores de resumo, como soma ou contagem, usando funções agregadas SQL, quando necessário.

Sintaxe SQL da Cláusula ORDER BY:

```
SELECT <colunas>
FROM <tabelas>
ORDER BY <coluna1> [ASC | DESC], <coluna2> [ASC | DESC], ...;
```

Cláusula HAVING

A cláusula **HAVING** é utilizada em conjunto com a cláusula **GROUP BY**, e permite especificar (filtrar) quais grupos de registros, resultantes da operação **GROUP BY**, serão exibidos na instrução **SELECT**, com base em condições específicas definidas na cláusula **HAVING**. Essa cláusula é particularmente útil quando você deseja aplicar filtros a grupos de dados agregados.

Sintaxe SQL da Cláusula HAVING:

```
SELECT <colunas>
FROM <tabelas>
GROUP BY <coluna(s)>
HAVING <condição>;
```

Funções Agregadas

As funções agregadas são ferramentas poderosas em SQL para realizar cálculos e resumir dados de um conjunto de registros em um único valor. Elas são úteis quando você deseja obter informações estatísticas ou realizar operações de resumo em seu banco de dados. As principais funções agregadas em SQL são: **SUM**, **AVG**, **COUNT**, **MIN** e **MAX**.

SUM: A função **SUM** é usada para calcular a soma de valores numéricos em um conjunto de dados. A função ignora valores nulos e funciona apenas com números.

Exemplo: Calcular o total de todas as Vendas

```
SELECT SUM(valor) AS total_vendas FROM Vendas
```

AVG: A função **AVG** calcula a média aritmética de valores numéricos em um conjunto de dados. A função ignora valores nulos e funciona apenas com números.

Exemplo: Calcular a média de idade dos Funcionários

```
SELECT AVG(idade) AS media_idade FROM Funcionarios;
```

COUNT: A função **COUNT** é usada para contar o número de registros em um conjunto de dados. Pode ser aplicada a qualquer tipo de dado, incluindo campos que podem conter valores nulos.

Exemplo: Contar a total de Alunos

```
SELECT COUNT(*) AS total_alunos FROM Alunos;
```

MIN e MAX: As funções **MIN** e **MAX** são usadas para encontrar o valor mínimo e máximo em um conjunto de valores, respectivamente. As funções podem operar em qualquer tipo de dado.

Exemplo: Selecionar a menor pontuação de todos os Jogadores

```
SELECT MIN(pontuacao) AS menor_pontuacao FROM Jogadores;
```

Exemplo: Selecionar o valor do maior salário dos Funcionários

```
SELECT MAX(salario) AS maior_salario FROM Funcionarios;
```

Funções agregadas com GROUP BY e HAVING

Em alguns casos, é preciso aplicar funções agregadas não apenas a um conjunto de registros, mas também a grupos desses registros. Isso é realizado usando a cláusula **GROUP BY**, onde os atributos são usados para formar grupos.

Exemplo: Calcular a média dos salários por Departamento

```
SELECT departamento, AVG(salario) AS media_salario FROM Funcionarios
GROUP BY departamento;
```

Para aplicar condições a grupos de registros em vez de aplicá-las a registros individuais, é utilizado a cláusula **HAVING**

Exemplo: Selecionar os departamentos com média salarial maior que 5000

```
SELECT departamento, AVG(salario) AS media_salario
FROM Funcionarios GROUP BY departamento
HAVING AVG(salario) > 5000;
```

4. BIBLIOTECA *PYTHON* PARA BANCO DE DADOS

As bibliotecas de conexão a banco de dados desempenham um papel fundamental no ecossistema de desenvolvimento *Python*, proporcionando uma ponte essencial entre as aplicações e os SGBDs. Com o crescente papel dos dados no panorama tecnológico contemporâneo, a habilidade de integrar de maneira eficaz aplicações *Python* a bancos de dados é muito importante para desenvolvedores, engenheiros de dados e cientistas da computação.

Ao entender as aplicações e diferenças dessas bibliotecas, os desenvolvedores podem tomar decisões sobre a escolha das ferramentas mais adequadas para seus projetos, considerando fatores como tipo de banco de dados, requisitos de desempenho e preferências de *design*.

Para as demandas tradicionais de banco de dados relacionais, existem bibliotecas que integram o*Python* com vários sistemas de banco de dados relacionais comumente utilizados, como *Sybase*, *Oracle*, *Informix*, *ODBC*, *MySQL*, *PostgreSQL*, *SQLite* entre outros. O mundo *Python* também definiu uma API - *Application Programming Interface* ou Interface de Programação de Aplicação de banco de dados portátil, para acessar sistemas de banco de dados SQL a partir de *scripts Python*, que se apresenta da mesma maneira em uma variedade de sistemas de banco de dados subjacentes.

Por exemplo, as interfaces do fornecedor implementam a API portátil, um *script* escrito para funcionar com o *MySQL* funcionará praticamente inalterado em outros sistemas (como o *Oracle* por exemplo); geralmente, você só precisa substituir a interface do fornecedor subjacente. O mecanismo de banco de dados *SQLite* é uma parte padrão do próprio *Python* desde a versão 2.5, suportando tanto a prototipagem quanto às necessidades básicas de armazenamento do programa.

Quanto aos bancos de dados *NoSQL* (não SQL), o módulo *pickle* [1] padrão do *Python* oferece um sistema simples de persistência de objetos, o qual permite que programas salvem e restaurem facilmente objetos *Python* completos em arquivos e objetos semelhantes a arquivos.

Na *Web*, é fácil se deparar com sistemas de terceiros de código aberto chamados *ZODB* e *Durus* que fornecem sistemas completos de banco de dados orientados a objetos para *scriptsPython*; outros, como *SQLObject* e *SQLAlchemy*, implementam mapeadores objeto-relacional (ORMs - *Object Relational Mappers*), que adaptam o modelo de classes do *Python* a tabelas relacionais; e ainda *PyMongo*, uma interface para o *MongoDB*, um banco de dados de documentos de estilo *JSON - JavaScript Object Notation* de código aberto e alto desempenho, que armazena dados em estruturas muito semelhantes às listas e dicionários nativos do *Python*, e cujo texto pode ser analisado e criado com o módulo *JSON* padrão do *Python*.

Outros sistemas ainda oferecem formas mais especializadas de armazenar dados, incluindo o *Datastore* no *Google App Engine*, que modela dados com classes *Python* e oferece escalabilidade extensiva, bem como opções adicionais de armazenamento em nuvem emergentes, como *Microsoft Azure*, *PiCloud*, *OpenStack* e *Stackato*.

---

[1] Módulo que traduz conveniente quase qualquer tipo de dado em uma *string*, para o seu posterior armazenamento em um banco de dados.

Bibliotecas de acesso ao *MySQL* em *Python*

Ao conectar *Python* a um banco de dados *MySQL*, a escolha da biblioteca certa desempenha um papel crucial. Dentre as opções disponíveis, destacam-se quatro principais: *MySQL Connector/Python*, *MySQLdb* (também conhecida como *MySQL-python*), *SQLAlchemy* e *DjangoORM*. Cada uma delas oferece um conjunto distinto de características e vantagens, adaptando-se a diferentes necessidades e preferências de desenvolvimento.

O *MySQL Connector/Python*, por exemplo, é a biblioteca oficial fornecida pela Oracle, oferecendo suporte completo para todas as versões do *MySQL*. Com uma API "*C*" pura e uma API *Python* pura, ele proporciona uma flexibilidade excepcional, incluindo suporte para autenticação de plugin e conformidade com o *DB API 2.0*.

Por outro lado, a *MySQLdb*, uma das bibliotecas mais antigas para conexão ao MySQL em *Python*, se destaca por sua interface de nível mais baixo e alta performance, impulsionada pela sua extensão em linguagem *C*. Embora ofereça uma conformidade total com o *DB API 2.0*, ela pode ter menos recursos em comparação com o *MySQL Connector/Python*.

Além dessas, temos o *SQLAlchemy*, uma ORM que permite o mapeamento de objetos *Python* para tabelas de banco de dados. Sua versatilidade é evidente no suporte a uma ampla gama de sistemas de banco de dados, não se restringindo apenas ao *MySQL*. O *SQLAlchemy* oferece dois modos de uso: Core (núcleo), que se baseia em SQL puro, e ORM, que segue uma abordagem orientada a objetos. Essa flexibilidade aliada ao suporte para transações e persistência de sessão o tornam uma escolha popular.

Por fim, o *Django ORM* é parte integrante do *framework Web Django*, sendo altamente integrado ao seu ecossistema. Com foco em facilitar o desenvolvimento rápido, ele oferece um sistema de migração automática de banco de dados, tornando-o uma opção sólida para projetos baseados em *Django*. No entanto, em termos de flexibilidade, pode ser um pouco menos abrangente em comparação com o *SQLAlchemy*.

Neste contexto, a escolha da biblioteca dependerá das necessidades específicas do projeto, bem como das preferências individuais de desenvolvimento. Cada uma dessas bibliotecas possui suas vantagens distintas, e a seleção adequada pode ser determinante para o sucesso da integração entre *Python* e *MySQL*.

Você Sabia?

A combinação de *Python* com SQL tornou-se tão comum que muitas bibliotecas, como *SQLAlchemy* e *DjangoORM*, foram criadas para simplificar e potencializar essa integração.

Esta parceria entre a linguagem de programação e a linguagem de consulta a banco de dados proporciona aos desenvolvedores uma maneira mais fluida e eficiente de interagir com bancos de dados, permitindo a criação de aplicações robustas e escaláveis.

Por meio destas bibliotecas, é possível não apenas executar consultas SQL diretamente em *Python*, mas também modelar e manipular estruturas de banco de dados usando *Python* como linguagem intermediária.

A biblioteca *mysql-connector-python*

A interação eficaz entre aplicações*Python* e bancos de dados *MySQL* é fundamental para o desenvolvimento de sistemas robustos e eficientes. Nesse contexto, a biblioteca oficial *MySQL*, **mysql-connector-python**, surge como uma ferramenta poderosa e versátil, projetada para simplificar e aprimorar a integração entre o ambiente *Python* e o SGBD *MySQL*.

Desenvolvida com foco na eficiência, facilidade de uso e conformidade com os padrões *MySQL*, a **mysql-connector-python** oferece uma ampla gama de recursos para atender às necessidades dos desenvolvedores. Desde a criação de conexões simples até operações avançadas de consulta e manipulação de dados, essa biblioteca se destaca pela sua adaptabilidade a diferentes cenários de desenvolvimento.

Instalação e Configuração

A utilização da biblioteca**mysql-connector-python** requer uma instalação adequada e configuração apropriada para garantir uma integração eficiente entre aplicações *Python* e bancos de dados *MySQL*. O primeiro passo é garantir que o *Python* e o seu gerenciador de pacotes ***pip*** estão instalados no sistema, em seguida, executar a seguinte linha de comando:

```
pip install mysql-connector-python
```

Após a instalação, é recomendável verificar se a biblioteca foi instalada corretamente. Isso pode ser feito importando a biblioteca no interpretador *Python*:

```
import mysql.connector
```

Para estabelecer a conexão entre o servidor *MySQL* e a linguagem *Python*, permitindo a manipulação das bases de dados no SGBD, os detalhes de conexão como o **host**, **usuário**, **senha** e o **nome** do Banco de Dados, devem ser configurados conforme pode ser visualizado no trecho de código a seguir.

```
import mysql.connector



# Configuração da conexão
conexao = mysql.connector.connect(
   host = "localhost",
   user = "seu_usuario",
   password = "sua_senha",
   database = "seu_banco_de_dados"
)
```

No código acima se instancia um conector e as configurações de conexão são passadas para o método **connect**, onde **host** é o IP do servidor (no caso servidor local), caso a instalação do *MySQL* esteja na mesma máquina em que se está programando podemos usar "**localhost**" ou o endereço IP "**127.0.0.1**", os campos **user** e **password** devem ser inseridos conforme foi configurado durante a instalação do *MySQL*, e **database** é o nome do banco de dados que se deseja manipular, uma vez que vários bancos podem ser utilizados no mesmo servidor.

Para se executar comandos SQL através da conexão configurada é necessário o uso de um **cursor**, que nada mais é que uma estrutura que permite a manipulação dos resultados de uma pesquisa de forma iterativa. O **cursor** atua como um ponteiro ou indicador que aponta para uma posição específica em um conjunto de resultados retornado por uma consulta, o comando abaixo exemplifica como é obtido um **cursor**.

```
cursor = conexao.cursor()
```

Com o **cursor**, é possível executar comandos SQL, como consultas, atualizações e inserções, utilizando métodos como **execute()**.

Sempre que uma operação for concluída, é importante realizar o **commit** para efetivar as alterações no banco de dados e fechar a conexão e o cursor para liberar os recursos que foram alocados e a sessão que foi aberta no servidor MySQL, o que se não for realizado pode gerar lentidão nas operações e até mesmo o travamento da aplicação. Para tal então excute o seguinte trecho de código.

```
conexao.commit()
cursor.close()
conexao.close()
```

Principais Funcionalidades

Utilizando o cursor obtido da conexão, os desenvolvedores podem executar comandos SQL de criação, leitura, atualização e exclusão de forma intuitiva através do uso da função **execute()**.

```
cursor = conexao.cursor()



# Exemplo de consulta
cursor.execute("SELECT * FROM tabela")



# Exemplo de inserção
cursor.execute("INSERT INTO tabela (coluna1, coluna2)
VALUES (%s, %s)", ("valor1", "valor2"))
```

No código acima foi executado um comando SQL de consulta diretamente, na segunda execução um comando de inserção é concatenado com variáveis previamente preenchidas, os valores a serem implementados na *query* (consulta em SQL) foram passados por parâmetro para a função, dessa forma antes de gravar no banco a *string*, as variáveis são sobrepostas para compor a *query* que será executada.

A biblioteca oferece também suporte a transações, permitindo que operações no banco de dados sejam agrupadas e confirmadas ou revertidas de uma só vez, como pode ser visualizado no trecho de código a seguir.

```
conexao.start_transaction()



# Operações SQL
conexao.commit()
```

Os resultados de consultas SQL podem ser recuperados de maneira fácil, seja em forma de tuplas, listas e dicionários.

```
# Recuperando resultados como tuplas
resultados = cursor.fetchall()
```

A biblioteca **mysql-connector-python** possui um sistema robusto de gerenciamento de exceções, o que facilita a detecção e o tratamento de erros durante operações no banco de dados.

```
try:

# Operações SQL
except mysql.connector.Error as erro:
print(f"Erro MySQL: {erro}")
```

Essas funcionalidades fazem da **mysql-connector-python** uma escolha sólida para desenvolvedores que buscam uma solução completa e eficiente para interagir com bancos de dados *MySQL* em seus projetos *Python*.

Exemplos Práticos

Como visto anteriormente, uma vez que se tem certeza que a biblioteca **mysql-connector-python** está instalada, deve-se primeiro configurar a conexão com o servidor *MySQL* local ou remoto, como pode ser observado na Figura 21.

```
import mysql.connector

conexao = mysql.connector.connect(
    user='apostila',
    password='AssassinsCreed@',
    host='127.0.0.1',
    database='LojaDeGames'
)

cursor = conexao.cursor()
```

Figura 21: Configuração da conexão

Tendo iniciado a conexão e o cursor para se executar as operações de criação, leitura, atualização e exclusão, basta utilizar os comandos **execute** e **commit**. O código a seguir apresenta a criação de uma tabela chamada **Jogos**.

```
query = "CREATE TABLE Jogos (id INT AUTO_INCREMENT PRIMARY KEY,
         nome VARCHAR(255) NOT NULL,"
query += " plataforma VARCHAR(50), data_lancamento DATE,
         descricao TEXT);"
cursor.execute(query)
conexao.commit()
```

Figura 22: Criação da tabela Jogos

A *query* descrita acima foi quebrada em duas partes simplesmente para melhorar a sua visualização, logo a seguir o comando é executado e as alterações aplicadas ao banco de dados.

O código abaixo apresenta a inserção de um jogo:

```
import datetime

lancamento = datetime.datetime(2017,11,13)

# Inserir um novo jogo
query = "INSERT INTO Jogos (nome, plataforma,
        data_lancamento, descricao)"
query += "VALUES (%s, %s)"
dados = ("Assassin's Creed", "XBOX 360", lancamento,
         "Assassin's Creed ...")
cursor.execute(query, dados)

# Certificar-se de fazer commit para salvar as alterações
conexao.commit()
```

Figura 23: Inserção de um jogo na tabela Jogos

Para se obter todos os jogos da tabela basta executar da mesma forma o comando SQL de **SELECT** junto da função **fetchall()** do cursor.

```
# Recuperar todos os Jogos
query = "SELECT * FROM Jogos"

cursor.execute(query)
jogos = cursor.fetchall()

for jogo in jogos:
    print(jogo)
```

Figura 24: Lista de todos os jogos da tabela Jogos

Para fazer uma alteração em um jogo basta executar o comando **UPDATE**, no caso do código abaixo é alterada a descrição de um jogo, logo em seguida o comando SQL é executado e as alterações são gravadas no banco de dados.

```
# Atualizar a descricao de um jogo
query = "UPDATE Jogos SET descricao =
         %s WHERE id = %s"
dados = ("O primeiro game da franquia ...", 1)

cursor.execute(query, dados)
conexao.commit()
```

Figura 25: Alteração dos dados de um jogo da tabela Jogos

No momento da exclusão de um registro deve-se ter um campo que o identifique de forma única, no código abaixo foi utilizado o ID do jogo "**jogo_id**", então basta a execução do comando SQL abaixo e do **commit** da conexão para que um registro seja removido do banco de dados. Mas cuidado, uma vez removido o registro não poderá ser recuperado.

```python
# Excluir um jogo
query = "DELETE FROM Jogos WHERE id = %s"
jogo_id = (1,)

cursor.execute(query, jogo_id)
conexao.commit()

cursor.close()
conexao.close()
```

Figura 26: Exclusão de um registro

Ao fim da execução dos comandos citados anteriormente basta fechar o cursor e a conexão.

PyMySQL

Em um cenário onde a integração eficiente entre aplicações *Python* e bancos de dados *MySQL* é essencial, a biblioteca ***PyMySQL*** se destaca como uma ferramenta poderosa e versátil. Desenvolvida para proporcionar uma interface fácil de usar e eficaz, o *PyMySQL* simplifica o processo de conexão, consulta e manipulação de dados em bancos *MySQL*, tornando-se uma escolha popular entre os desenvolvedores.

Ela é projetada para ser uma implementação pura em *Python* do protocolo *MySQL*, o que significa que não requer a instalação de módulos adicionais ou bibliotecas nativas. A biblioteca oferece uma maneira fácil e eficiente de se comunicar com servidores *MySQL* usando Python.

Ao compreender as nuances do *PyMySQL*, os desenvolvedores estarão aptos a otimizar com eficiência suas aplicações, garantindo uma integração tranquila e segura com bancos de dados *MySQL*.

Instalação e Configuração

A instalação e configuração adequadas da biblioteca *PyMySQL* são passos cruciais para estabelecer uma conexão eficiente entre aplicações *Python* e bancos de dados *MySQL*. Assim como a biblioteca apresentada anteriormente, esta biblioteca pode ser instalada através do gerenciador de pacotes ***pip***, caso o *Python* e o ***pip*** já estejam instalados o seguinte comando deve ser executado:

```
pip install pymysql
```

Após a instalação, é recomendável verificar se a biblioteca foi instalada corretamente. Isso pode ser feito importando a biblioteca no interpretador Python:

```
import pymysql
```

A forma de configurar a biblioteca ***PyMySQL*** é muito parecida com a biblioteca ***MySQL-connector-python***, onde deve-se passar o **IP** do servidor, **usuário**, **senha** e **banco de dados** que se deseja fazer a conexão.

```python
import pymysql

# Configuração da conexão
conexao = pymysql.connect(
    host="localhost",
    user="seu_usuario",
    password="sua_senha",
    database="seu_banco_de_dados"
)
```

Figura 27: Conexão com a biblioteca *PyMySQL*

Um **cursor** é necessário para executar comandos SQL na conexão. Após a criação da conexão, o cursor deve ser obtido da mesma forma que a biblioteca anterior através da função **cursor()**. Com o cursor, é possível executar comandos SQL, como consultas, atualizações e inserções, utilizando métodos como **execute()**.

Uma vez que uma conexão e um cursor foram instanciados ambos devem ser fechados ao final de sua utilização, mas não antes de aplicar as alterações no banco de dados utilizando o **commit()**, este procedimento é idêntico à biblioteca apresentada anteriormente.

```python
conexao.commit()
cursor.close()
conexao.close()
```

Ao seguir esses passos, o programador estará pronto para utilizar a biblioteca *PyMySQL* em seus projetos *Python*, facilitando a conexão e manipulação eficaz de dados em bancos de dados *MySQL*.

Principais Funcionalidades

A biblioteca *PyMySQL* oferece uma gama de funcionalidades robustas e eficientes para facilitar a integração entre aplicações Python e bancos de dados *MySQL*. A forma de se utilizar a biblioteca é muito semelhante à anterior, uma vez que a conexão foi configurada deve-se obter um cursor para se executar os comandos SQL, como consultas, atualizações e inserções.

```python
cursor = conexao.cursor()

# Exemplo de consulta
cursor.execute("SELECT * FROM tabela")

# Exemplo de inserção
cursor.execute("INSERT INTO tabela (coluna1, coluna2) VALUES (%s, %s)", ("valor1", "valor2"))
```

O código acima é idêntico ao utilizado na biblioteca anterior, o que diferencia uma execução da outra é a conexão que foi configurada para se obter o cursor. Sendo assim os resultados de consultas SQL podem ser recuperados através de atribuição de maneira fácil, seja em forma de tuplas, listas e dicionários.

```python
# Recuperando resultados como tuplas
resultados = cursor.fetchall()
```

O *PyMySQL* oferece também suporte a transações, permitindo que operações no banco de dados sejam agrupadas e confirmadas ou revertidas de uma só vez, a forma de se utilizar é idêntica à biblioteca anterior.A biblioteca possui um sistema sólido de gerenciamento de exceções, facilitando a detecção e o tratamento de erros durante operações no banco de dados.

```
try:
    # Operações SQL
except pymysql.Error as erro:
    print(f"Erro PyMySQL: {erro}")
```

Figura 28: Tratamento de exceção com a biblioteca *PyMySQL*

Desenvolvedores podem ajustar parâmetros de conexão, como a autenticação do **cliente**, para atender a requisitos específicos de segurança.

```
conexao = pymysql.connect(
    host="localhost",
    user="seu_usuario",
    password="sua_senha",
    database="seu_banco_de_dados",
    client_flag=pymysql.constants.CLIENT.SSL
)
```

Figura 29: Inserção de parâmetro de conexão com autenticação

Essas funcionalidades tornam o *PyMySQL* uma ferramenta versátil e poderosa para desenvolvedores que buscam uma solução eficaz para interagir com bancos de dados *MySQL* em seus projetos *Python*.

Exemplos Práticos

Uma vez que se tem certeza que a biblioteca *PyMySQL* está instalada deve-se primeiro configurar a conexão com o servidor *MySQL* local ou remoto:

```
import pymysql.connector

conexao = pymysql.connector.connect(
    user='apostila',
    password='AssassinsCreed@',
    host='127.0.0.1',
    database='LojaDeGames'
)

cursor = conexao.cursor()
```

Figura 30: Configuração da conexão

Todos os códigos apresentados abaixo serão idênticos aos usados na biblioteca **mysql-connector-python**, uma vez que basta alterar o **import** e a configuração da conexão para que ele funcione corretamente.

Tendo iniciado a conexão e o cursor para se executar as operações de criação, leitura, atualização e exclusão, basta utilizar os comandos execute e commit. O código a seguir apresenta a criação de uma tabela chamada Jogos.

```
query = "CREATE TABLE Jogos (id INT AUTO_INCREMENT PRIMARY KEY,
         nome VARCHAR(255) NOT NULL,"
query += " plataforma VARCHAR(50), data_lancamento DATE,
          descricao TEXT);"
cursor.execute(query)
conexao.commit()
```

Figura 31: Criação da tabela Jogos

A *query* descrita acima foi quebrada em duas partes simplesmente para melhorar a sua visualização, logo a seguir o comando é executado e as alterações aplicadas ao banco de dados.

O código abaixo apresenta a inserção de um jogo:

```
import datetime

lancamento = datetime.datetime(2017,11,13)

# Inserir um novo jogo
query = "INSERT INTO Jogos (nome, plataforma,
         data_lancamento, descricao)"
query += "VALUES (%s, %s)"
dados = ("Assassin's Creed", "XBOX 360", lancamento,
          "Assassin's Creed ...")
cursor.execute(query, dados)

# Certificar-se de fazer commit para salvar as alterações
conexao.commit()
```

Figura 32: Inserção de um jogo na tabela Jogos

Para se obter todos os jogos basta executar da mesma forma o comando SQL de **SELECT** junto da função **fetchall()** do cursor, da mesma forma como realizado na Figura 24.

Para fazer uma alteração em um jogo basta executar o comando **UPDATE**, da mesma forma como no caso do código na Figura 25, que altera a descrição de um jogo, que em seguida executa o comando SQL e as alterações são então gravadas no banco de dados.

No momento da exclusão de um registro deve-se ter um campo que o identifique de forma única, no código abaixo foi utilizado o ID do jogo, basta tal basta a execução do comando SQL como demonstrado na Figura 26 com o comando **DELETE**, para que um registro seja removido do banco de dados. Mais uma vez alertamos para o cuidado a ser tomado neste tipo de execução, pois uma vez removido o registro, o mesmo não poderá ser recuperado.

Ao fim da execução dos comandos demonstrados anteriormente basta fechar o cursor e a conexão. Valendo destacar que os códigos foram executados com as duas bibliotecas, e o seu comportamento foi o mesmo para ambas.

SQLAlchemy

A biblioteca ***SQLAlchemy*** é uma ferramenta poderosa e flexível para o mapeamento objetos-relacionais (ORM) em *Python*. Projetada para simplificar a interação entre aplicações *Python* e banco de dados relacionais, como *MySQL*, *PostgreSQL* e *SQLite*. Ela facilita a interação com bancos de dados SQL usando uma abordagem orientada a objetos (OO),

além de oferecer suporte a SQL puro quando necessário. O uso dessa biblioteca permite que a troca de SGBD seja feita mais facilmente, uma vez que basta trocar algumas linhas de código para que a conexão e as requisições sejam executadas.

Diferentemente de abordagens tradicionais de interação com bancos de dados, onde SQL é escrito manualmente, a *SQLAlchemy* utiliza modelos de dados *Python*, fornecendo uma interface intuitiva e OO. Essa abstração facilita a criação, leitura, atualização e exclusão CRUD [1] - *Create, Read, Update and Delete* de registros no banco de dados, tornando o processo mais natural para desenvolvedores familiarizados com programação OO.

Apesar de oferecer uma camada de abstração em nível de objeto, a *SQLAlchemy* permite a expressividade total da linguagem SQL quando necessário. Isso proporciona flexibilidade para lidar com consultas complexas e personalizadas.

A biblioteca facilita a definição de relacionamentos entre tabelas, refletindo a estrutura relacional do banco de dados. Os desenvolvedores podem especificar associações entre modelos, simplificando consultas que envolvem múltiplas tabelas.

SQLAlchemy gerencia automaticamente sessões e transações, simplificando o controle de operações no banco de dados. As sessões proporcionam um contexto para trabalhar com objetos e garantem a atomicidade nas transações.

A *SQLAlchemy* é compatível com uma variedade de SGBDs, oferecendo aos desenvolvedores a flexibilidade de escolher a solução que melhor se adequa aos requisitos do seu projeto. A sua arquitetura modular permite estender suas funcionalidades ou personalizar comportamentos específicos conforme necessário. Isso é especialmente útil em casos em que é necessário ajustar o comportamento padrão da biblioteca.

Ao adotar a *SQLAlchemy*, os desenvolvedores podem usufruir de uma poderosa ferramenta que simplifica a interação com bancos de dados relacionais, promovendo boas práticas de programação e proporcionando um ambiente mais produtivo para o desenvolvimento de aplicações *Python* robustas e orientadas a dados.

---

[1] É um mnemônico para as quatro operações (Criar, Ler, Alterar e Apagar) básicas de armazenamento persistente, referentes ao conjunto de ações que podem ser realizadas em “bancos/bases de dados”.

Instalação e Configuração

Para se utilizar a *SQLAlchemy* é necessário se certificar de que ela foi devidamente instalada, além disso, você precisará da biblioteca que de fato fará a conexão do *Python* com o *MySQL*, o **mysql-connector-python** ou **pymysql** são duas opções mais comuns. O ***pip*** pode ser usado para instalar o *SQLAlchemy* e a biblioteca de conexão escolhida, como fizemos anteriormente, porém é claro, alterando-se para a biblioteca em questão.

```
pip install sqlalchemy
```

Após a instalação, você pode verificar se a *SQLAlchemy* foi instalada corretamente importando-a no interpretador Python:

```
import sqlalchemy
```

Use a função **create_engine** do *SQLAlchemy* para criar uma instância do motor de banco de dados, especificando o tipo de banco de dados, **usuário**, **senha**, **host** e **nome** do banco de dados:

```
from sqlalchemy import create_engine
# substitua 'seu_usuario', 'sua_senha', 'seu_host' e 'seu_bd' pelos seus próprios detalhes
```

```
engine = create_engine('mysql+mysqlconnector://seu_usuario:sua_senha@seu_host/seu_bd')
```

Substitua **'mysql+mysqlconnector'** pelo o do seu driver *MySQL*, como por exemplo **'mysql+pymysql'** se você estiver usando o *PyMySQL*.

Na *SQLAlchemy*, os modelos são representados por classes *Python*. Cada instância da classe corresponde a uma linha em uma tabela do banco de dados. No exemplo a seguir, definem-se modelos criando classes que herdam da classe **Base** e incluindo atributos que representam colunas no banco de dados.

```python
from sqlalchemy import Column, Integer, String, Sequence
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Pessoa(Base):
    __tablename__ = 'pessoas'
    id = Column(Integer,
    Sequence('pessoa_id_seq'), primary_key=True)
    nome = Column(String(50))
    idade = Column(Integer)
```

Figura 33: Definição de modelos herdados da classe Base

Uma vez definidos os modelos, você pode criar o banco de dados e as tabelas associadas utilizando o método **create_all()**.

```
Base.metadata.create_all(engine)
```

Principais Funcionalidades

SQLAlchemy fornece uma camada de mapeamento objeto-relacional que permite aos desenvolvedores trabalhar com objetos *Python* em vez de escrever SQL diretamente. As tabelas do banco de dados são representadas por classes *Python*, simplificando a interação com o banco de dados.

Uma vez que o modelo foi criado para manipular a base de dados basta instanciar um objeto da classe e adicionar o objeto a sessão, a alteração será gravada no banco de dados após a execução do comando **commit()**.

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from seu_modulo import Pessoa, Base

engine = create_engine('sqlite:///exemplo.db')
Base.metadata.bind = engine

DBSession = sessionmaker(bind=engine)
session = DBSession()

# Criar um novo registro
nova_pessoa = Pessoa(nome='João', idade=25)
session.add(nova_pessoa)
session.commit()
```

Figura 34: Mapeamento de objeto-relacional

Apesar de oferecer uma camada de abstração em nível de objeto, *SQLAlchemy* permite que os desenvolvedores escrevam consultas SQL expressivas quando necessário. Isso é especialmente útil para consultas complexas ou personalizadas. Além de fornecer suporte para migrações de banco de dados, permitindo que você evolua o esquema do banco de dados de maneira controlada ao longo do tempo.

A arquitetura modular da *SQLAlchemy* permite que você estenda suas funcionalidades ou customize comportamentos específicos conforme necessário. Isso é útil em casos em que é necessário ajustar o comportamento padrão da biblioteca. Gerencia automaticamente o *pooling*[1] de conexões, o que pode melhorar significativamente o desempenho em ambientes de aplicativos com vários usuários concorrentes.

A *SQLAlchemy* é frequentemente utilizada em conjunto com o *framework Flask* <https://flask.palletsprojects.com/> para desenvolvimento *Web*, proporcionando uma integração fácil e eficiente para construção de aplicativos robustos.

---

[1] É um cache de conexões de banco de dados que são compartilhadas e reutilizadas, o que melhora o desempenho e a latência das conexões com os bancos.

Exemplos Práticos

Para que a *SQLAlchemy* seja utilizada para atender requisições do *MySQL*, é necessário que uma das bibliotecas apresentadas anteriormente estejam instaladas, além do próprio *SQLAlchemy*, uma vez que que se tenha as bibliotecas basta iniciar o programa com a configuração da conexão com o banco de dados, como demonstrado a seguir.

```
from sqlalchemy import create_engine, Column, Integer, Text, Date, String, Sequence
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from datetime import datetime

conexao = "mysql+mysqlconnector://apostila:AssassinsCreed2017@localhost/LojaDeGames"

engine = create_engine(conexao, echo=True)

Base = declarative_base()
Session = sessionmaker(bind=engine)
```

Para que um objeto possa ser instanciado e posteriormente gravado no banco de dados é necessária a criação de uma classe.

```
from sqlalchemy import Column, Integer, Text, Date, String, Sequence

class Jogo(Base):
    __tablename__ = 'Jogos'
    id = Column(Integer, Sequence('jogo_id_seq'), primary_key=True)
    nome = Column(String(255))
    plataforma = Column(String(50))
    data_lancamento = Column(Date())
    descricao = Column(Text(255))
```

Após a implementação da classe que irá instanciar os nossos jogos, basta criar um objeto desta classe e uma sessão, para que o objeto seja persistido no banco de dados.

from datetime import datetime

```
# Criar um novo jogo
nome = "Assassin's Creed"
plataforma = "XBOX 360"
data_lancamento = datetime("2017","11","13")
descricao = ""
novo = Jogo(nome,plataforma,data_lancamento,descricao)

# Iniciar uma sessão
session = Session()

# Adicionar e confirmar o novo jogo
session.add(novo)
session.commit()
```

A recuperação dos objetos do banco de dados é feita de forma simples, basta pegar os objetos da sessão que foi aberta com o método **query()**.

```
# Recuperar todos os Jogos
jogos = session.query(Jogo).all()

for jogo in jogos:
print(jogo.id, jogo.nome, jogo.descricao)
```

Para editar uma linha do banco de dados basta buscar um objeto por um de seus atributos, sobrescrever a variável e dar um **commit** no objeto.

```
# Atualizar a descrição de um jogo
jogo = session.query(Jogo).filter(id=1).first()
jogo.descricao = "Assassin's Creed foi o primeiro jogo da franquia ..."

# Confirmar Atualização
session.commit()
```

A remoção é feita de forma similar, uma vez que primeiro se filtra o objeto a ser removido e logo em seguida o comando de remoção é criado por causa do método **delete()**, para que a remoção seja efetuada é necessário usar o método **commit()**. Ao fim de todas as manipulações é necessário que a sessão com o banco de dados seja fechada através do método **close()**.

```
# Excluir um jogo
jogo = session.query(Jogo).filter_by(id=1).first()
session.delete(jogo)

# Confirmar Exclusão
session.commit()
session.close()
```

Django

Django é um *framework Web* de alto nível, escrito em *Python*, que encoraja o desenvolvimento rápido e limpo de aplicações *Web* robustas e escaláveis. Criado por desenvolvedores experientes, o *Django* segue o princípio do "*Don't Repeat Yourself*" (**DRY**) ou algo como “Não se repita” o que favorece a abordagem do "*batteries-included*" ou “baterias

incluídas", fazendo alusão de que a linguagem *Python* oferece um conjunto abrangente de ferramentas e bibliotecas prontas para uso, ou seja, tudo o que você precisa.

Possui ORM integrado, o que facilita a interação com bancos de dados relacionais. Modelos *Python* são utilizados para definir a estrutura do banco de dados, proporcionando uma abstração eficaz e evitando a necessidade de escrever SQL manualmente.

Um painel de administração pronto para uso é gerado automaticamente para modelos definidos, permitindo a administração fácil e eficiente de dados. Isso é especialmente útil para tarefas administrativas comuns. *Django* utiliza um sistema de rotas para mapear URLs para funções de visualização (*views*). Isso permite criar páginas *Web* dinâmicas e responder a diferentes solicitações de forma modular e organizada.

O sistema de *templates* (modelos prontos) do *Django* permite a criação de páginas dinâmicas de forma simples e elegante. Esses *templates* facilitam a mistura de HTML com código *Python*, proporcionando flexibilidade e reusabilidade. Inclui um sistema de autenticação integrado que lida com funções comuns, como *login*, *logout* e recuperação de senhas. Além disso, oferece uma estrutura de autorização baseada em permissões.

Django oferece suporte para a criação de APIs de forma fácil e consistente, permitindo que suas aplicações interajam com outras aplicações e serviços. Um ponto positivo do *framework* é a sua comunidade ativa de desenvolvedores e usuários que contribuem regularmente para o aprimoramento do mesmo. A abundância de recursos, tutoriais e documentação torna o aprendizado e o desenvolvimento com *Django* acessíveis.

Instalação e Configuração

A instalação e configuração do *Django* são relativamente simples e seguem uma abordagem padrão para ambientes *Python*. Como descrito anteriormente, certifique-se de ter o *Python* instalado em seu sistema. Recomenda-se o uso do *Python 3.x.*, assim como as outras bibliotecas apresentadas, o *Django* deve ser instalado via ***pip***, como demonstrado a seguir.

```
django-admin --version
```

Para criar um novo projeto *Django* o comando **startproject** deve ser utilizado, isso criará uma estrutura de diretórios para o seu projeto *Django*, o nome do projeto é o fornecido após **'startproject'**.

```
django-admin startproject nome_do_projeto
```

Para inicializar o servidor de desenvolvimento basta acessar o diretório do projeto e executar o comando **runserver** a partir do arquivo **manage.py** localizado na raiz do projeto.

```
python manage.py runserver
```

Isso iniciará o servidor em **http://127.0.0.1:8000/**. Abra então um navegador *Web* e acesse esse endereço para acessar a página inicial do *Django*. Vale ressaltar que a biblioteca de acesso ao banco de dados deve ser instalada antes da execução da aplicação *Django*, uma vez que a *engine* irá utilizar funções da biblioteca selecionada para fazer a conexão com o banco de dados.

O *Django* utiliza um banco de dados por padrão. Para configurar um banco de dados, vá para o arquivo **settings.py** dentro do diretório do seu projeto e ajuste as configurações relacionadas ao banco de dados, como o **tipo** de banco de dados, **nome** do banco de dados, **usuário** e **senha**, como mostrado na Figura 35

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'seu_banco_de_dados',
        'USER': 'seu_usuario',
        'PASSWORD': 'sua_senha',
        'HOST': 'local_host',
        'PORT': '5432',
    },
```

Figura 35: Configurações relacionadas ao banco de dados em Django

Após configurar o banco de dados, execute o comando **migrate** para aplicar as migrações e criar as tabelas:

```
python manage.py migrate
```

Principais Funcionalidades

Uma das principais funcionalidades do *Django* é o seu ORM integrado, o que facilita a interação com bancos de dados relacionais. Isso permite que você defina modelos *Python* para representar suas tabelas de banco de dados, simplificando a manipulação de dados.

O ponto de partida para se trabalhar com o *ORM Django* é definir modelos *Python* que representam as tabelas do banco de dados. Cada modelo é uma classe que herda da classe **django.db.models.Model**, como pode ser visualizado na Figura 36.

```python
from django.db import models

class Pessoa(models.Model):
    nome = models.CharField(max_length=100)
    idade = models.IntegerField()
```

Figura 36: Definição de modelos de tabelas em Django

Os campos no modelo representam colunas no banco de dados. O *Django* oferece vários tipos de campos, como **CharField**, **IntegerField**, **DateField**, para mais informações acesse a documentação oficial do *framework* em <https://docs.djangoproject.com/en/4.2/>.

Depois de definir modelos, você precisa criar migrações, que nada mais são que *scripts Python* que descrevem as alterações no banco de dados. Para este fim deve-se usar o comando **makemigrations** e logo em seguida aplicar este *script* ao banco de dados através do comando **migrate**, como demonstrado a seguir.

```
python manage.py makemigrations

python manage.py migrate
```

Django fornece um conjunto rico de métodos de QuerySet (conjunto de busca que na essência é uma lista de objetos de um dado modelo), para recuperar dados de bancos de dados de maneira eficiente através de objetos.

```python
# Criar uma nova pessoa
nova_pessoa = Pessoa(nome='João', idade=25)
nova_pessoa.save()

# Atualizar uma pessoa existente
pessoa = Pessoa.objects.get(id=1)
pessoa.idade = 30
pessoa.save()

# Excluir uma pessoa
pessoa = Pessoa.objects.get(id=1)
pessoa.delete()
```

Figura 37: Recuperação de dados de um banco em Django

Há recursos *Django* para que estas operações possam ser feitas diretamente pelo SQL utilizando a função **raw()**, mas geralmente estas tarefas são executadas por funções de objetos como **objects.all()**, **objects.filter()**, essa manipulação direta através de objetos torna a manipulação do banco de dados mais fácil, permitindo que o desenvolvedor alcance seus objetivos mesmo que ele não tenha um vasto conhecimento de SQL

O *Django* suporta vários tipos de relacionamentos entre modelos, como **ForeignKey**, **OneToOneField** e **ManyToManyField**, como demonstrado na Figura 38.

```python
class Livro(models.Model):
    titulo = models.CharField(max_length=200)
    autor = models.ForeignKey(Autor, on_delete=models.CASCADE)
```

Figura 38: Relacionamento de chave estrangeira em Django

Exemplos Práticos

A primeira coisa que precisamos ter certeza é que temos o *Django* é uma das bibliotecas **mysql-connector-python** ou *PyMySQL* instaladas, logo em seguida um projeto deve ser criado com o comando **startproject**, como demonstrado a seguir, para a criação do projeto **apostila**.

```
python-admin startproject apostila
```

No arquivo **settings.py** do seu projeto *Django*, configure as informações do banco de dados para apontar para o *MySQL*. Logo em seguida o comando para se criar um App deve ser executado para que a estrutura de arquivos seja criada, como demonstrado na Figura 39.

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'LojaDeGames',
        'USER': 'apostila',
        'PASSWORD': 'AssassinsCreed@',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

Figura 39: Informações de banco de dados em Django para apontar para o MySQL

Uma vez que um novo App foi criado dentro do arquivo **models.py**, este deve ser editado e todos os modelos devem ser gravados no banco de dados, como demonstrado na Figura 40.

```
from django.db import models

class Jogo(models.Model):
    nome = models.CharField(max_length=255)
    plataforma = models.CharField(max_length=50)
    data_lancamento = models.DateField()
    descricao = models.CharField(max_length=255)
```

Figura 40: Gravação de modelos em Django

Após a criação dos modelos os comandos **makemigrations** e **migrate** devem ser executados para que o modelo seja criado no banco de dados. Para manipular os objetos desta classe basta instancia-la dentro do documento **views.py**, como demonstrado da Figura 41.

```
def listaJogos():
    return Jogo.objects.all()

def addJogo():
    nome = "Assassin's Creed"
    plataforma = "XBOX 360"
    data = datetime.datetime("2017","11","13")
    descricao = "O primeiro jogo da franquia que arrebatou..."
    jogo = Jogo(nome=nome, plataforma=plataforma, data=data,
                descricao=descricao)
    jogo.save()
    return True
```

Figura 41: Manipulação de objetos em Django

Para alterar ou remover objetos basta usar o a função **get_object_or_404** e fornecer o modelo o identificador do registro, esta função retorna somente um registro, para efetivar a alteração no banco de dados basta usar o método **save()** e **delete()** do objeto, como demonstrado na Figura 42.

```
def updateJogo(jogo_id):
    jogo = get_object_or_404(Jogo, id=jogo_id)
    jogo.descricao = "Assassin's Creed é um jogo de aventura..."
    jogo.save()

def deleteJogo(jogo_id):
    jogo = get_object_or_404(Jogo, id=jogo_id)
    jogo.delete()
```

Figura 42: Alteração e Remoção de objetos em Django

Você já percebeu que o *Django* pode oferecer muito mais do que vimos até aqui, e se fossemos explorar todos os seus recursos, teríamos que dedicar um curso bastante extenso para demonstrar em detalhes suas potencialidades. Como isso não é possível, sugerimos que você busque por mais informações sobre o funcionamento do *framework* no *website* oficial do mesmo em <https://www.djangoproject.com/>, aproveite também para fazer o seu *download* do, instale-o, configure-o e teste não só os recursos mostrados aqui, mas também realize seus próprios.

Chegamos ao final de nosso curso, ficou claro que o universo dos Bancos de Dados e bastante extenso, então procure acessar todos os sites indicados e fique atento(a) às dicas e informações extras que disponibilizamos ao longo do mesmo. Pesquise e se desafie, pois dessa maneira rapidamente estará desenvolvendo suas próprias aplicações de maneira correta e eficaz, o que certamente lhe abrirá inúmeras boas oportunidades profissionais.

Parabéns por chegar até aqui!

Tags do conteúdo

Modelo Coceitual, Modelo Lógico , Modelo Físico

SGBDs, SGBDs, Sistema de Gerenciamento de Banco de Dados

SQL, *Python*

MySQL, PyMySQL, Django

Biblioteca, Conexão, Persistênicia, OO, Orientação a Objetos

MER, ER, Modelo Entidade-Relacionamento

Referências

ALEXANDRUK, Marcos. **Modelagem de banco de dados**. São José dos Campos, SP, 2011.

DOCPLAYER. **APOSTILA DE SQL SERVER 7.0**. 2017. Disponível em: <https://docplayer.com.br/52850840-Apostila-de-sql-server-7-0.html>. Acesso em: 06 out. 2023

ELMASRI, R.; NAVATHE, S. B. **Sistemas de banco de dados**. Tradução de Daniel Vieira; revisão técnica de Enzo Seraphim e Thatyana de Faria Piola Seraphim. 6. ed. São Paulo: Pearson Addison Wesley, 2011. Título original: Fundamentals of database systems. ISBN 978-85-4301-381-7.

QUINTÃO, P**. Banco de Dados Relacionais - Parte I**. Livro Eletrônico. [S.l.: s.n.], 2020.

SOUZA, Warley. **Banco de Dados - Parte 2**. 2019. Disponível em: <https://medium.com/@warleysoares35/banco-de-dados-5649e2cd2dc8>. Acesso em: 04 out. 2023.

TEOREY, T. J.; LIGHTSTONE, S.; NADEAU, T. **Projeto e Modelagem de Banco de Dados**. Tradução de Daniel Vieira. Rio de Janeiro: Elsevier, 2007. Tradução de: Database Modeling and Design (4th ed). ISBN 85-352-2114-X.

TETILA, E. C. **Banco de Dados Relacional**. 1. ed. Curitiba: Appris Editora, 2022. ISBN 9786525005980.

VICENTE, A. T. O. **Mapeamento, Conversão e Migração automática de Bancos de Dados Relacionais para Orientados a Grafos**. 2020. 97 p. Dissertação (Mestrado em Ciência e Tecnologia da Computação) - Universidade Federal de Itajubá, Itajubá, MG, 2020.