Integração de MYSQL e Python

**Índice**

1. **Introdução, Requisitos e Ferramentas Utilizadas**
   1. SQL e Python
   2. Requisitos
   3. Ferramentas Utilizadas
2. **Python**
   1. Setup
   2. Importar e Instalar Módulos
   3. Uso de Variáveis
      1. Declaração de Variáveis
      2. Input e Print
      3. Comando If/Else e Indentações
      4. Listas e Índice
   4. Loops While e For
   5. Exceções
   6. Criar e Chamar Funções
3. **SQL**
   1. Setup
   2. Planejamento do Banco de Dados
      1. Objetivo do Banco de Dados
      2. Tables, Atributos e Relações entre Tables
   3. CREATE, DROP, SELECT e ALTER Tables
   4. INSERT, UPDATE e DELETE Tables
   5. QUERIES
   6. CREATE e CALL Procedures
4. **Integração Entre Python e MySQL**
   1. Definição das Funções Principais
   2. Execução de Queries
   3. Código de Exemplo Explicado

**Introdução, Requisitos e Ferramentas Utilizadas**

**1.1 - SQL e Python**

O objetivo dessa apostila é ensinar ao aluno como realizar a integração entre a linguagem Python e a linguagem SQL, e para isso é interessante explicar o que é cada uma dessas linguagens, seus usos e a razão de querermos realizar essa integração.

Python é uma das linguagens de programação mais populares e mais utilizadas atualmente, oferecendo uma enorme versatilidade, devido em parte ao grande número de módulos e pacotes que foram criados por desenvolvedores ao longo dos anos, que nos dão um grande número de opções para realizar qualquer função que quisermos. Essa versatilidade fornecida pelos módulos é exatamente o que nos permitirá mais tarde nessa apostila realizar a conexão entre o Python e SQL que desejamos.

A Structured Language Query, ou Linguagem de Consulta Estruturada, que chamamos de SQL, é uma linguagem de programação criada com o propósito específico de controlar e gerenciar bancos de dados relacionais. Isto é, bancos de dados que guardam não só os dados, mas também contém a informação de como esses dados se relacionam entre si. SQL é muito utilizado na área de análise de dados, e permite que uma pessoa armazene e acesse uma enorme quantidade de informação com facilidade realizando Queries (Consultas).

Com essa informação em mente, fica claro a razão de querermos realizar a integração das duas linguagens. Com a versatilidade do Python e a quantidade de informação que os bancos de dados relacionais oferecem, podemos criar, organizar e automatizar processos extremamente complexos e detalhados, ou podemos utilizar Python para criar uma interface gráfica, que permitiria que até uma pessoa sem nenhum conhecimento de SQL acessasse e modificasse seguramente as informações nos bancos de dados.

**1.2 - Requisitos**

Em termos de hardware o único requisito necessário é um computador capaz de rodar o Python e o servidor SQL.

Em termos de software, todos os programas utilizados nessa apostila são gratuitos ou open-source, portanto não haverá nenhum custo monetário, e os links para encontrar os programas recomendados podem ser encontradados na seção Anexo. O PC utilizado durante esse processo está usando o Windows 10 como OS, então se o aluno utilizar Mac ou Linux podem haver diferenças em alguns aspectos, portanto recomendo se possível utilizar o Windows durante o acompanhamento dessa apostila.

Em termos de conhecimentos prévios, nenhum é necessário, essa apostila foi criada com a intenção de que qualquer pessoa pudesse acompanhar o processo do ínicio ao fim, e aprender a utilizar o Python, SQL, e como combiná-los. Dito isso, algum conhecimento básico sobre programação seria útil para entender mais rapidamente cada parte do assunto, mas não é necessário. Na seção de Anexos você pode encontrar alguns materiais que podem auxiliar no aprendizado dos conceitos mais básicos, caso a explicação fornecida aqui não seja suficiente para o entendimento.

Por fim, essas linguagens foram feitas com a língua inglesa em mente, portanto inevitavelmente partes da sintaxe utilizadas por elas estarão em inglês. Tentarei manter o uso de inglês ao mínimo, e quando não for possível evitá-lo tentarei traduzir ou explicar o significado. Ainda sim um conhecimento básico de inglês facilitaria a compreensão.

**1.3 - Ferramentas Utilizadas**

No caso do Python, utilizaremos a versão 3.11, e como IDE utilizaremos a versão gratuita do PyCharm. Se você já tiver experiência com o Python, use o que você achar mais confortável, mas essa será a configuração utilizada para todos os exemplos apresentados nessa apostila, logo será mais fácil de acompanhar se você também a utilizar.

No caso do SQL precisamos não só escolher o programa que nós usaremos, mas também o tipo de SQL. O SQL não é uma linguagem única, existem várias versões com suas próprias características. No nosso caso, utilizaremos o MySQL pois é uma das versões mais populares, além de ser gratuita e open-source.

Além disso precisaremos de uma interface gráfica para facilitar o processo de escrever e organizar nossos queries. Eu recomendo o uso de duas interfaces diferentes: o PopSQL será nossa interface primária, pois ela é mais clara e fácil de entender, e o MySQL workbench será usado ocasionalmente pois ele nos permite utilizar alguns comandos específicos do MySQL que não funcionam corretamente no PopSQL.

A última coisa que precisamos é instalar um servidor SQL no PC que usaremos. Esse é um processo um pouco mais complexo do que instalar um programa normal, mas também é fácil, e será explicado detalhadamente na seção **Setup** da parte sobre SQL.

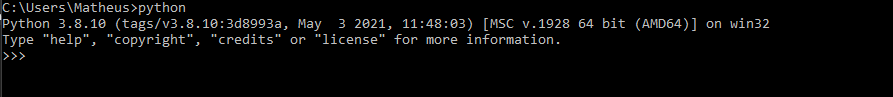
Como mencionado anteriormente, os links para o download de todos esses programas podem ser encontrados na seção **Anexos**.

**Python**

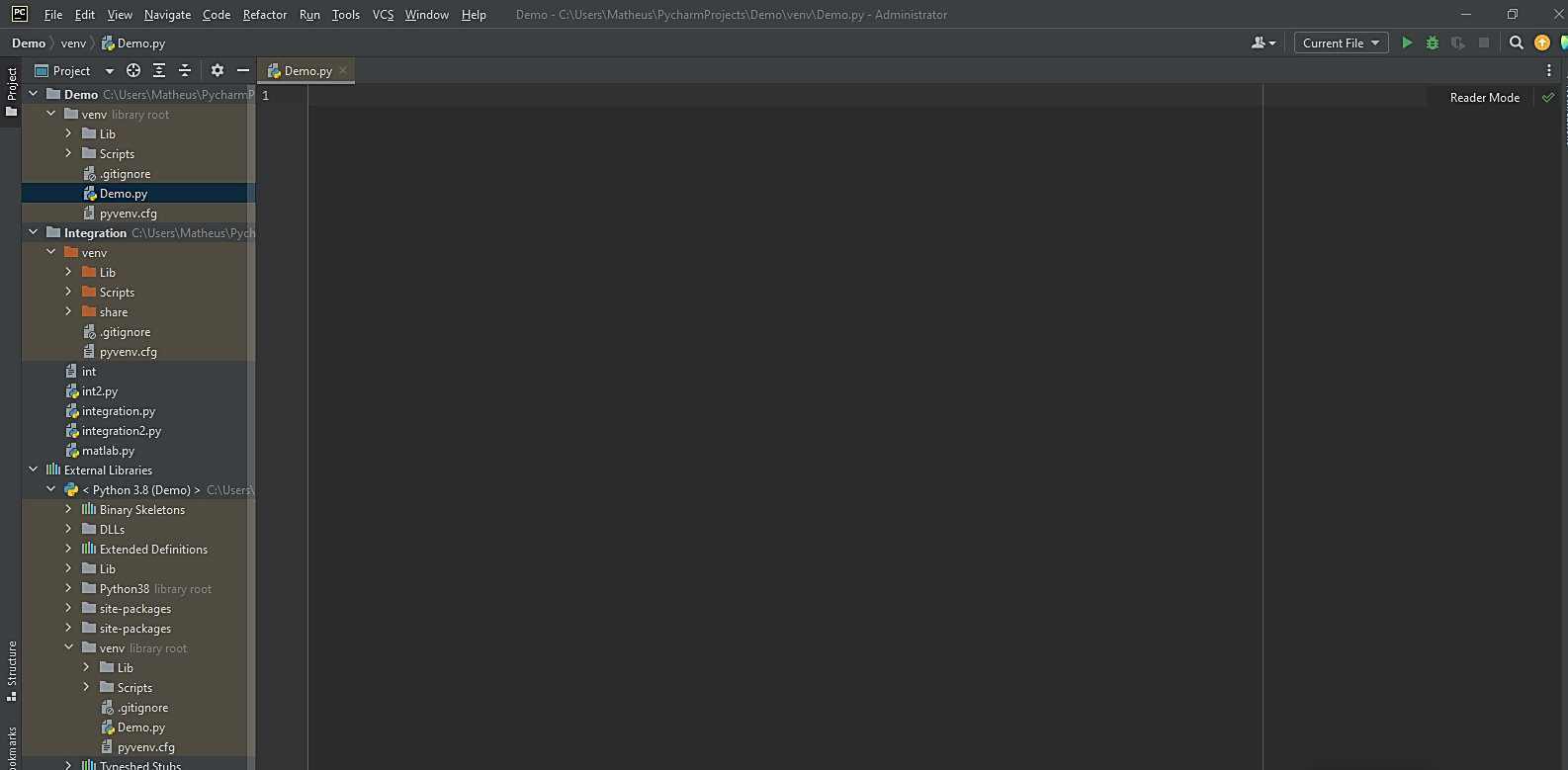
**2.1 - Setup**

Faça o download da versão mais recente do python, e da versão mais recente do PyCharm Community. Realize a instalação dos dois programas. Depois disso, abra o Prompt de Comando do seu computador e digite Python.

Deve aparecer uma indicação de que você acessou o python e qual a versão do python está instalada. Se sua tela estiver como na imagem seguinte, o python está instalado, você pode testar digitando uma conta básica como 2+2 e ele deve te dar a resposta.



Agora abra o PyCharm, e inicie um novo projeto. Se certifique de que o interpreter selecionado seja a versão do python que está instalada no seu computador. Depois que o projeto estiver criado, crie um novo arquivo do tipo python. Se sua tela estiver assim, então podemos começar a aprender o necessário sobre Python.

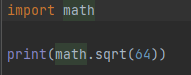


**2.2 - Importar e Instalar Módulos**

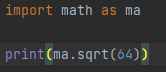
Como havia sido mencionado anteriormente, em python podemos utilizar módulos, e eles são muito importantes pois eles nos dão acesso a muitas funções extremamente úteis. Para exemplificar tente rodar o seguinte código:



Ao tentar rodar, deve ter ocorrido um erro que diz que math não está definido. Isso ocorre porque nós ainda não importamos o módulo math. Para fazer isso simplesmente digite no topo do código import math.



Agora ao rodar o programa, devemos obter o valor da raíz quadrada de 64, pois é isso que a função math.sqrt faz. Para usarmos uma função de um módulo geralmente escrevemos o nome do módulo, ponto, e o nome da função desejada. Caso não desejemos digitar o nome todo do módulo toda vez que usarmos uma de suas funções podemos escrever:

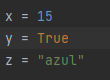


Agora onde normalmente escreveríamos math podemos escrever apenas ma. Você pode escolher qualquer abreviação que desejar, só tome cuidado para não utilizar a mesma abreviação para dois módulos diferentes.

**2.3 - Uso de Variáveis**

**2.3.1 - Declaração de Variáveis**

Para declararmos uma variável, simplesmente precisamos escolher o nome da variável e escolher o valor que queremos para ela:

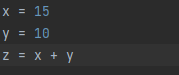


X tem o valor de 10 e é do tipo inteiro. Y tem o valor True e é do tipo boolean, ou seja, só tem duas opções de valor, verdadeiro(true) ou falso(false). Z é uma string, que contém o valor azul. Para declarar algo como uma string precisamos utilizar as aspas, e o valor da variável será o que estiver entre as aspas.

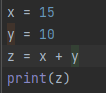
**2.3.2 - Print e Input**

Print e input são dois comandos muito importantes para possibilitar que o usuário interaja com o programa.

No caso do print ele é utilizado toda vez que quisermos que o programa escreva algo que nós possamos ler, ou seja, sem o print o programa irá rodar e não nos dará nenhum feedback sobre seu funcionamento, como no caso do exemplo seguinte:

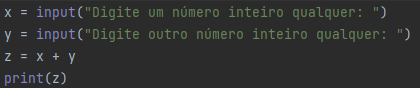


Se quisermos que o python nos diga o valor de z, precisaremos adicionar o comando print dessa forma:



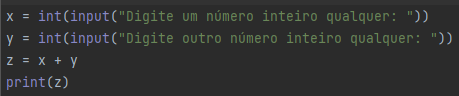
Então devemos receber o valor esperado de 25.

No caso do comando input ele é utilizado para que o usuário possa introduzir valores ao programa em momentos relevantes enquanto o código está rodando. Veja o código a seguir:



Na primeira linha, o programa irá escrever a frase que aparece entre os parênteses, irá ficar esperando até que o usuário digite algo, e então atribuirá o valor introduzido pelo usuário a variável x. O mesmo ocorre na segunda linha, para a variável y, e na quarta linha recebemos o valor da sua soma.

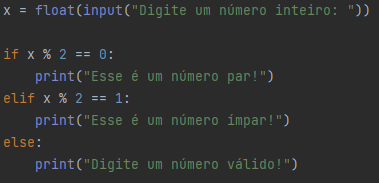
Entretanto se você tentar rodar esse programa irá perceber que ele não está funcionando corretamente. Se colocar x = 1 e y = 2, você receberá 12 como resultado. Isso ocorre porque o programa interpretou os números digitados nas primeiras linhas como strings e não como inteiros. Para consertar isso basta escrever dessa forma:



Esse comando int() converte os valores que estão dentro dos seus parênteses para valores inteiros se possível. É importante sempre prestar atenção aos tipos de variáveis e valores que estão presentes no programa, para evitar esses tipos de erros. Ao rodar o programa ele deve funcionar corretamente agora.

**2.3.3 - Comando If e Indentação**

O comando if é um dos aspectos mais importantes para programação em geral, especialmente em um nível mais básico, e é o comando que nos permite impor condições ao programa, que irá passar a se comportar de forma diferente dependendo dessas condições.



Vamos analisar esse programa. Na primeira linha nós pedimos para o usuário digitar um número, e então convertemos essa string para o tipo float, que permite números decimais, e guardamos esse valor na variável x. Então chegamos no primeiro If, onde vemos um novo operador, o %, que significa realizar a divisão do primeiro número pelo segundo e guardar apenas o resto dessa divisão, isto é, 5 % 2 seria igual a 1, ou 4 % 2 seria igual a zero.

Podemos perceber que nesse tipo de divisão, para qualquer valor par de x o resultado será 0 e para qualquer valor ímpar de x o resultado será 1, ou seja, se x % 2 = 0 sabemos que x é par, e se x % 2 = 1 sabemos que x é ímpar.

Assim podemos ver que no primeiro if, se x for um número par, o código que está imediatamente abaixo dele será executado, e nesse caso irá escrever para o usuário que o número é par. Caso não seja par, o código irá pular tudo que está diretamente abaixo do primeiro if, nos levando ao elif.

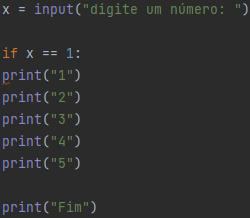
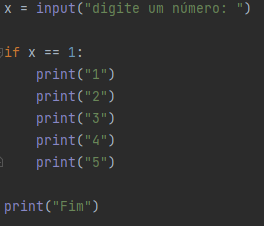
Isso quer dizer else if, e indica que esse segundo if está relacionado ao primeiro, ou seja, só será checado caso a condição do primeiro if não seja verdadeira. Então o programa irá realizar a divisão para ver se o número é ímpar, e se for irá executar o código diretamente abaixo do elif, que irá escrever para o usuário que o número é ímpar.

Caso nenhuma das condições anteriores seja satisfeita o código irá sempre então executar o que estiver diretamente abaixo do else, nesse caso irá pedir que o usuário escolha um número válido. Você pode testar isso fornecendo um número decimal ao programa.

O if sempre será checado primeiro, seguido de quaisquer elif que apareçam, e se nenhuma das condições tiver sido satisfeita, o else sempre será executado. Vale a pena lembrar, que não é sempre necessário colocar um elif ou um else, o if pode aparecer sozinho, e pode utilizar qualquer condição desde que os únicos valores possíveis para essa condição sejam True e False.

Outro ponto importante é que quando queremos checar se uma igualdade é verdadeira utilizamos == ao invés de =, pois o == checa a igualdade dos valores enquanto o = atribui um valor.

O uso de if é nosso primeiro encontro com um aspecto importante de python, a indentação. A indentação é um aspecto frequente em várias linguagens de programação, porém enquanto na maioria delas a indentação é algo meramente estético que ajuda apenas a trazer mais clareza na leitura do código, no python a indentação é a forma utilizada para determinar a ordem de execução do código. Vejamos o exemplo dessas duas imagens:

Essas duas imagens são parecidas, mas a da esquerda apresentará um erro, enquanto a da direita executará o código corretamente. Isso ocorre porque na primeira todas as linhas de código estão no mesmo nível de indentação, ou seja, o python lê o programa como se não tivesse nada sendo executado pelo if, enquanto que na segunda imagem os comandos print estão indentados corretamente e estão sendo considerados “dentro” do if, logo serão executados se a condição dada for satisfeita.

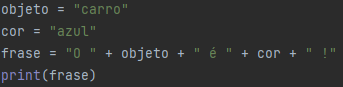
Podemos perceber que o último print não está indentado, logo ele ocorre independente do comando if. Ao testar o código veremos que o “Fim” sempre será escrito, enquanto a contagem de 1 a 5 só será escrita se x inserido pelo usuário for de fato igual a 1.

Logo é importante, lembrar de indentar corretamente os elementos de acordo com a ordem de execução desejada, e é importante que a indentação seja uniforme para cada nível da ordem de execução, isto é, se você quer que um comando seja executado consecutivamente ao outro eles devem estar no mesmo nível de indentação. Para mantermos as endentações uniformes podemos utilizar a tecla tab, que sempre adiciona o mesmo número de espaços.

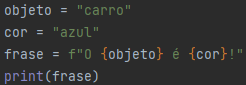
**2.3.4 - Operações com Strings**

Para podermos executar comandos de SQL através de python precisaremos saber como utilizar strings. Enquanto os detalhes serão vistos mais na frente, veremos agora o básico necessário para isso.

Como vimos anteriormente, podemos somar duas strings para combiná-las em uma. Isso é chamado de concatenação de strings, e o melhor método para realizar isso é a f-string.

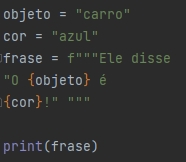


Nessa imagem vemos o método normal, no qual para formarmos a frase “O carro é azul!”, para que a palavra carro, e a palavra azul possam ser mudadas simplesmente alterando a variável, precisamos recortar a frase para poder montá-la corretamente, o que se torna cada vez mais imprático conforme a string que queremos escrever vai aumentando.



Nessa imagem vemos a chamada f-string, que pode ser usada colocando a letra f antes das aspas no ínicio da string e colocando o nome da variável desejada entre chaves dentro da string. Dessa forma independente do tamanho da string ou do número de variáveis, a concatenação de strings pode ser feita sem perda de clareza na sua leitura.

Outro recurso útil no trabalho com strings são as aspas triplas. Elas nos permitem controlar melhor a formatação da string ao escrevê-la ao longo de múltiplas linhas, além de nos permitir adicionar aspas normais dentro da string.



Se rodarmos esse programa vamos ver que a frase “ Ele disse ‘ O carro é azul!’ ” vai ser escrita ao longo de 3 linhas, exatamente como especificamos, e as aspas dentro da string foram interpretadas corretamente como parte da string ao invés de como o início/fim da string. Ambas essas características serão úteis quando quisermos juntar o python e o SQL.

**2.3.5 – Listas e Índice**

Em python uma variável pode ser um grupo de múltiplos valores, que chamamos de uma lista. Essa lista pode ser composta de quantos elementos nós quisermos, e cada um desses elementos dentro da lista pode ser representado por um índice. A contagem do índice sempre irá iniciar em zero, ou seja, o primeiro elemento da lista tem índice zero, o segundo tem índice 1, e assim por diante.

Podemos também adicionar ou remover elementos dessa lista livremente, ou apenas checar individualmente cada um desses elementos. Vamos ver todos esses processos no programa a seguir:

cores = ["vermelho" , "azul", "verde"]  
print(cores)  
print(cores[1])  
  
nova\_cor\_1 = input("Digite o nome de uma cor: ")  
cores.append(nova\_cor\_1)  
print(cores)  
nova\_cor\_2 = input("Digite o nome de outra cor: ")  
indice\_nova\_cor\_2 = int(input("""Em qual posicao da lista deve entrar essa nova cor: """))  
cores.insert(indice\_nova\_cor\_2,nova\_cor\_2)  
  
print(cores)  
  
del\_cor\_1 = input("Digite o nome da cor a ser retirada da lista:")  
cores.remove(del\_cor\_1)  
  
print(cores)  
  
del\_pos = int(input("Remover a cor que esta em qual posicao:"))  
del cores[del\_pos]  
  
print(cores)

Na primeira linha nós definimos o nome da lista como cores e os 3 elementos que fazem parte dela. Em seguida nos escrevemos a lista para ver se ela está correta, e vemos como escrever um elemento específico da lista, nesse caso o elemento de índice 1, ou seja, o segundo elemento da lista. Para isso basta utilizar o nome da lista seguido do índice dentro de colchetes como visto no exemplo acima.

Em seguida começamos a alterar a lista, pedindo que o usuário insira uma nova cor na lista. Esse novo valor é guardado em uma variável e em seguida utilizamos o método append para adicionar o novo valor na lista. O método append é escrito da seguinte forma:

*nome\_da\_lista.append(valor\_a\_ser\_inserido)*

Esse comando irá inserir o valor desejado diretamente ao final da lista. A lista será então escrita de novo para verificarmos a alteração.

O próximo método, chamado de insert, pede o nome de uma cor e o índice no qual se deve ser inserido esse valor. O método é escrito da seguinte forma:

*nome\_da\_lista.insert(índice\_desejado, valor\_desejado)*

Esse comando irá inserir o valor desejado na posição do índice desejado. Novamente escrevemos a lista para verificar a alteração.

Agora mostraremos como remover elementos. O método remove é escrito da seguinte forma:

*nome\_da\_lista.remove(valor\_a\_ser\_removido)*

Isso irá remover qualquer elemento da lista que tenha valor igual ao fornecido no comando. Mais uma vez escrevemos a lista para observar o processo.

Por fim o método del, escrito como:

*del nome\_da\_lista[índice\_desejado]*

Isso nos permite deletar o elemento associado ao índice especificado. Por fim escrevemos uma última vez a lista, observando as mudanças ocorridas.

**2.4 - Loops While e For**

Quando criamos um programa muitas vezes queremos repetir uma mesma ação por um tempo determinado, ou realizar um determinado número de iterações, ou simplesmente manter o programa rodando constantemente, e para isso precisamos utilizar os comandos While e For.

O comando While repete todo o código que está indentado dentro dele, enquanto uma determinada condição se mantém satisfeita. Caso essa condição deixe de ser verdadeira, o loop é encerrado.

O comando For repete o código indentado dentro dele um certo número X predeterminado de vezes, uma vez que se repita X vezes o loop é encerrado.

while True:  
  
 x = float(input("Digite um número inteiro: "))  
  
 if x % 2 == 0:  
 print("Esse é um número par!")  
 elif x % 2 == 1:  
 print("Esse é um número ímpar!")  
 else:  
 print("Digite um número válido!")

Nesse programa, fazemos o mesmo processo de checar se um número fornecido pelo usuário é par ou ímpar, porém com um While que mantém o programa se repetindo indefinidamente. Agora não é mais necessário reiniciar o programa para inserir outro número, o comando while True sempre é verdadeiro, logo o loop nunca será encerrado.

Caso desejemos encerrar o loop em algum momento só precisamos criar uma condição que pode passar a se tornar falsa, como no exemplo a seguir:

y = True  
while y == True:  
  
 x = float(input("Digite um número inteiro: "))  
  
 if x % 2 == 0:  
 print("Esse é um número par!")  
 elif x % 2 == 1:  
 print("Esse é um número ímpar!")  
 else:  
 print("Digite um número válido!")  
 y = False

Agora o loop só se repete enquanto y tiver o valor True, e quando o número não é par ou ímpar o valor False é atribuído ao y, encerrando o loop.

O For é comumente utilizado quando queremos iterar ao longo de uma listo ou algum outro conjunto de valores. Veja os exemplos a seguir:

cores = ["vermelho","azul","verde","amarelo","roxo","branco","preto"]  
  
for x in cores:  
 print(x)  
  
for x in range(3):  
 print(cores[x])

Nesse programa temos uma lista contendo várias cores, e dois loops for. No primeiro loop, ele está programado para repetir uma vez para cada x presente na lista “cores”. Esse x indica um elemento qualquer, e funciona como uma variável que tem valor igual ao primeiro elemento da lista durante o primeiro ciclo, igual ao segundo elemento no segundo ciclo e assim em diante. Esse loop será repetido uma vez para cada elemento dentro da lista cores, e no final resultará em escrevendo a lista inteira, um elemento de cada vez.

O segundo loop é similar, com a principal diferença sendo o número de repetições e a interação da variável x. Range(3) significa todos os números de 0 até 3, gerando uma lista que contém os números 0, 1 e 2. Portanto o loop se repete 3 vezes, primeiro com x = 0, depois x = 1 e por fim x = 2. Entretanto agora o x está indicando o índice dentro da lista cores, ou seja, essa forma nos permite selecionar apenas os 3 primeiros elementos da lista cores ao invés de termos que iterar ao longo da lista inteira.

**2.5 – Exceções**

Quando criamos um programa geralmente queremos que ele possa funcionar continuamente sem precisarmos reiniciá-lo constantemente. Porém podem haver erros que causam o encerramento do nosso programa de forma indesejada. Para lidar com isso podemos utilizar exceções, através do comando Try/Except.

while True:  
  
 x = float(input("Digite um número inteiro: "))  
  
 if x % 2 == 0:  
 print("Esse é um número par!")  
 elif x % 2 == 1:  
 print("Esse é um número ímpar!")  
 else:  
 print("Digite um número válido!")

Vamos retornar a esse programa que usamos anteriormente. Ele funciona continuamente graças ao comando while, porém se tentarmos digitar uma palavra ao invés de um número durante o input, ocorrerá um erro que encerrará o programa. Para resolvermos esse problema basta utilizarmos exceções.

while True:  
 try:  
 x = float(input("Digite um número inteiro: "))  
  
 if x % 2 == 0:  
 print("Esse é um número par!")  
 elif x % 2 == 1:  
 print("Esse é um número ímpar!")  
 else:  
 print("Digite um número válido!")  
 except:  
 print("Digite um número, não uma palavra!")

Agora quando executarmos o programa ao invés do erro que encerraria o programa, o programa vai apenas escrever a frase “Digite um número, não uma palavra!”. Isso ocorre pois quando usamos try/except, o programa irá tentar executar todo o código que está indentado dentro do comando try. Caso ocorra um erro, o programa para de executar o código dentro de “try” e executa o código dentro de “except”, que nesse caso apenas escreve uma mensagem.

Enquanto isso é extremamente útil, podem ocorrer problemas se usarmos exceções de forma irresponsável. Nesse caso, a nossa mensagem será exibida não importa o tipo de erro que ocorra, o que significa que nosso usuário pode estar digitando um número e não uma palavra, e nosso programa estará comunicando corretamente ao usuário qual o verdadeiro problema que está encerrando o programa.

Para resolvermos esse problema precisamos especificar mais a nossa exceção, para isso remova o try/except, e cause o erro de novo. No texto do erro, estará escrito o tipo de erro que causou o fechamento do programa, que nesse caso é um ValueError. Então precisaremos apenas fazer a seguinte modificação:

while True:  
 try:  
 x = float(input("Digite um número inteiro: "))  
  
 if x % 2 == 0:  
 print("Esse é um número par!")  
 elif x % 2 == 1:  
 print("Esse é um número ímpar!")  
 else:  
 print("Digite um número válido!")  
 except ValueError:  
 print("Digite um número, não uma palavra!")

Agora nossa mensagem só será mostrada quando esse tipo específico de erro ocorrer, nos permitindo identificar corretamente qualquer outro erro que possa ocorrer.

**2.6 – Criar e chamar funções**

Muitas vezes enquanto estamos programando nós repetimos as mesmas porções de código, que dependendo do tamanho dessas porções pode deixar o programa muito mais difícil de ler, além da possibilidade de cometermos erros quando estamos repetindo essas seções do código.

Para resolver isso podemos criar funções, que funcionam como uma porção de código que nós só precisamos escrever uma vez, e depois disso podemos chamar essas funções facilmente quando quisermos executar essa porção de código novamente. Vejamos o exemplo a seguir:

def elevar\_quadrado(num1):  
 y = pow(num1,2)  
 print(y)  
  
  
while True:  
 x = float(input("Digite um número para elevar ao quadrado: "))  
 elevar\_quadrado(x)

Na primeira linha antes no loop while nós definimos a função elevar\_quadrado, que pega o número entre parênteses, insere o número na função pow(num1, 2), no lugar de num1, e escreve o resultado dessa operação.

Dentro do loop, o usuário digita um número que é atribuído a variável x, que é então colocado dentro da nossa função elevar\_quadrado(num1) no lugar de num1. Agora sempre que quisermos elevar um número ao quadrado basta digitar elevar\_quadrado(número\_desejado).

Esse é um exemplo bem simples, mas as funções podem ser tão complicadas quanto quisermos, e quando estivermos utilizando o SQL em conjunto com python usaremos funções frequentemente. Isso nos permitirá criar um código bem mais legível, e bem mais compacto pois todas as seções repetidas do código podem ser substituídas com uma única linha de código.

**SQL**

**3.1 - Setup**

Para podermos começar a trabalhar com o MySQL, precisamos primeiro instalar o servidor SQL no PC que estamos usando. Para isso baixe os programas encontrados nos links da seção de Anexos.

Abra o instalador do mysql, e vá para a seção de adicionar componentes à instalação. Nós queremos instalar 4 componentes específicos, o MySQL server, o MySQL Workbench, o MySQL Shell, e o Connector/ODBC. O MySQL server está na seção de mesmo nome, o Workbench e o Shell estão na seção Applications, e o Connector está na seção MySQL Connectors.

Escolha a versão mais atual de cada componente e depois de selecioná-los vá para a próxima janela, espere o download dos componentes acabar, e siga para janela de instalação.

Após essa instalação, o programa iniciará a configuração do nosso servidor. Escolha o tipo do servidor como Standalone e continue, na seção de networking verifique se o Config Type está como Development Computer e prossiga, escolha a opção de Strong Password e crie um password para o seu servidor. Guarde bem essa senha, pois será necessária sempre que quisermos nos conectar ao servidor. Depois disso, selecione a opção “Start the MySQL server at system startup”, e prossiga até o final da instalação.

Agora para testar se o servidor foi criado corretamente, vamos tentar criar um banco de dados. Procure nos seus programas pelo MySQL Command Line Client e execute o programa, que então deve pedir a senha que criamos anteriormente. Depois de inserir a senha deve aparecer um cursor no qual podemos executar comandos de SQL. Vamos criar um banco de dados para testarmos, e podemos fazer isso digitando o comando: CREATE DATABASE teste;

Note que é importante utilizarmos o ponto e vírgula no final do comando porque ele indica o final de um comando de SQL. Além disso o nome do banco de dados pode ser o que quisermos, nesse caso dei o nome de “teste” ao banco, e as palavras “create” e “database” não precisam necessariamente ser escritas em letra maiúscula, mas é um bom hábito para se criar pois nos ajuda a diferenciar entre um comando e um nome de tabela ou de variável.

Se estiver tudo funcionando corretamente devemos receber um retorno de que o query foi executado com sucesso, e então podemos selecionar nosso banco de dados. Escreva o comando “USE teste;” e se recebermos a mensagem “Database Changed”, podemos então fechar o programa.

Agora só precisamos baixar o PopSQL, instalá-lo e executar o programa. No menu lateral esquerdo, seleciona a aba Schemas. No topo você deve ver o servidor, e abaixo dele o nome do banco de dados, então podemos nos conectar ao servidor e ao banco de dados correto. Para isso selecione a opção de servidor, e adicione um novo servidor.

De o nome que quiser para a conexão, no hostname/port coloque localhost e 3306, em database digite o nome do banco de dados que queremos usar, em usuário escreva root e use o password que criamos anteriormente, e aperte o botão connect.

**3.2 – Planejamento do Banco de Dados**

**3.2.1 – Objetivo do Banco de Dados**

Não existe uma forma objetivamente correta de se estruturar um banco de dados, porque nenhum banco de dados existe em um vácuo. Cada banco de dados será diferente, devido ao objetivo específico para o qual ele foi criado, e às necessidades as quais que ele deve atender.

Por isso antes de começar a criar nosso DB(banco de dados), precisamos saber exatamente qual é o objetivo dele, as informações que ele deve guardar, as relações que ele deve fazer, e o máximo possível de detalhes sobre ele. Para aprendermos SQL vamos então montar, passo-a-passo, um banco de dados para uma biblioteca, que será descrito em mais detalhes no parágrafo a seguir:

*A biblioteca deverá ser pública, e ter um número indeterminado de filiais que são todas parte do mesmo sistema, além de serem todas físicas, sem componente online. Os usuários poderão se cadastrar no sistema da biblioteca, e uma vez cadastrados, pegar emprestado livros em qualquer das filiais, sem custo monetário. Cada usuário poderá pegar emprestado, no máximo, 3 livros simultaneamente, e terão um prazo de devolução de até 30 dias após a retirada do livro. Os livros serão fornecidos a biblioteca, através de contato direto entre ela e as editoras que os publicam. Cada filial terá seus próprios funcionários, mas um funcionário pode trabalhar em mais de uma filial simultaneamente.*

Esse parágrafo nos dá uma boa ideia do funcionamento dessa biblioteca e nos dá muitas informações importantes para a criação do DB. Em muitos casos quando construímos um DB, a pessoa que pede o banco não é necessariamente a pessoa que irá criar esse DB, logo é preciso que o objetivo seja extremamente claro, para que o criador do banco de dados saiba o que ele pode ou não pode fazer.

Por exemplo, nesse texto foi especificado que não existe componente online nessas filiais, o que é importante pois significa que nosso sistema só precisa considerar livros físicos e portanto, precisa de uma forma de diferenciar entre cópias do mesmo livro, além de lidar com cópias finitas de cada livro, que não seriam necessárias se os empréstimos fossem digitais.

Apesar disso, existem muitos aspectos que foram propositalmente não especificados nesse texto, então precisamos usar nosso raciocínio e nossos conhecimentos para criar um DB que atenda essas especificações, e ao mesmo tempo esteja o mais completo possível.

**3.2.2 – Tables, Atributos e Relações entre Tables**

Um banco de dados relacional é composto basicamente de várias tabelas, que chamaremos de Tables, que estão conectadas entre si, e cada tabela armazena uma variedade de informações. Logo o primeiro passo para criar um DB, é determinar quais tabelas precisaremos criar, ou seja, quais são os aspectos mais importantes desse sistema que precisam ter suas informações armazenadas.

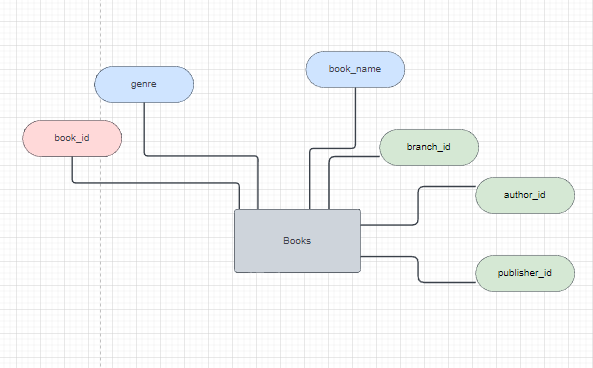
Temos uma biblioteca, que tem o objetivo de emprestar livros para usuários registrados no sistema. Então precisamos armazenar informação sobre esses usuários cadastrados, sobre os livros que estão na biblioteca, e sobre quais livros estão emprestados. Isso nos dá nossas 3 primeiras tables: Livro, Usuário, e Empréstimo.

Sabemos que temos várias filiais, que precisamos de funcionários em cada uma dessas filiais, e sabemos que para obter os livros precisamos entrar em contato com as editoras diretamente. Isso nos dá nossas próximas 3 tables: Filial, Funcionário, e Editora.

Precisaremos de mais duas tables, que não parecem ser necessárias inicialmente, mas que terão sua importância explicada em detalhe mais a frente: Emprego, e Autor.

Agora que temos nossa lista de tables vamos ver quais informações serão armazenadas em cada uma delas, e o porquê.

Livro:



Esse é um diagrama que representa a tabela livros, onde podemos ver quais atributos dos livros nós queremos armazenar. Vamos começar pelos atributos em azul, que são os atributos comuns:

- book\_name: Nome do Livro, essa é a coluna na qual armazenaremos o nome pelo qual o livro foi publicado. Essa é a forma mais fácil de identificar um livro, e é uma informação vital para o nosso sistema.

- genre: Gênero, essa é a coluna na qual armazenaremos o estilo do livro, se ele é de fantasia, ou ficção científica, romance, etc. Informação útil para os usuários poderem escolher os livros que os interessam, além de facilitar o processo de dividir os livros em seções.

Agora vamos ver os atributos em verde:

- branch\_id: Identidade da Filial, em qual das filiais da biblioteca esse livro está localizado. Com essa informação podemos saber quais filiais tem quais livros, que é útil para garantirmos uma variedade boa de livros em cada filial, ou para que o usuário possa saber para onde ir para achar o livro que está procurando.

- author\_id: Identidade do Autor, indica qual é o autor do livro em questão, útil para classificar o livro e para que o usuário encontre livros que se adequem aos seus interesses.

- publisher\_id: Identidade da Editora, indica qual é a editora que publicou o livro, útil para sabermos quais livros são de cada editora já que a biblioteca negocia diretamente com as editoras pelos livros, e útil para que o usuário encontre uma edição específica de determinado livro.

Por último vamos ver o atributo em vermelho:

- book\_id: Identidade do Livro.

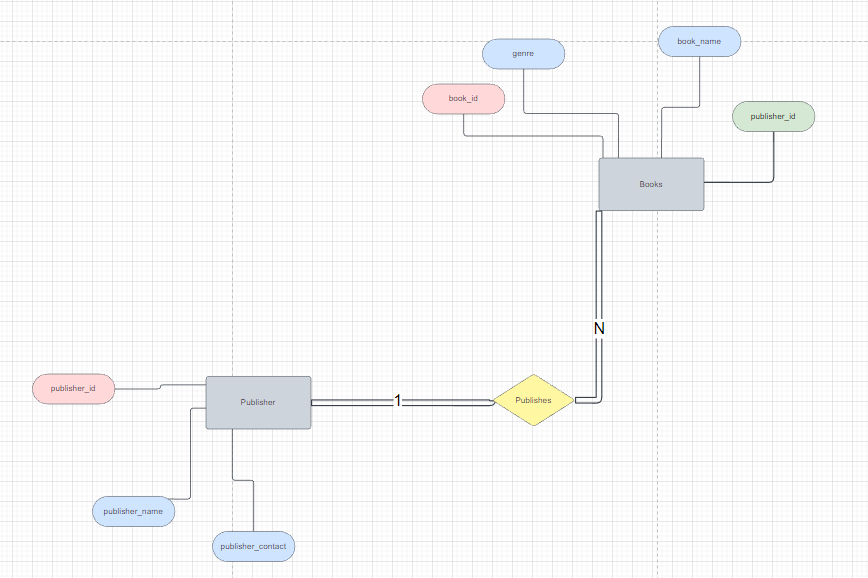
Essa coluna é o que chamamos de Primary Key(PK), ou seja, nossa coluna primária. Toda table em SQL precisa ter uma PK, e nenhum valor nessa coluna pode ser nulo, ou igual ao outro. A PK deve ser a forma de identificar unicamente cada linha da tabela.

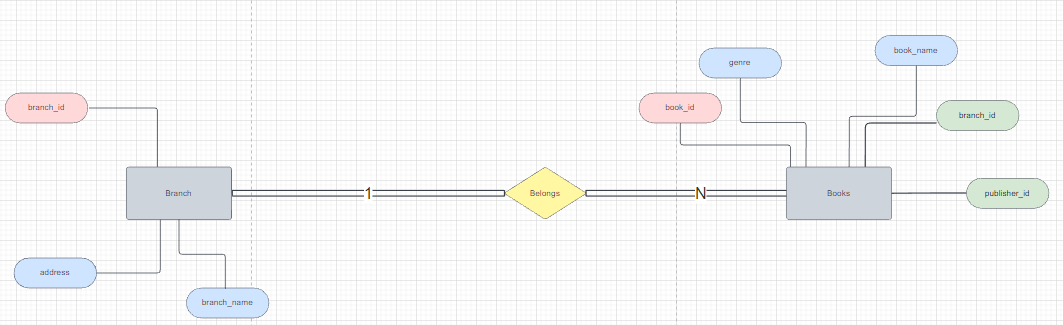
Por exemplo, digamos que nós tenhamos duas cópias de “O Conde de Monte Cristo”, essas duas cópias terão o mesmo nome, o mesmo autor, o mesmo gênero, possivelmente vão estar na mesma filial e provavelmente terão sido publicadas pela mesma editora. Isso significa que seria impossível diferenciar essas duas cópias no nosso DB, com os atributos que temos no momento. Para isso criamos esse atributo de Identidade do Livro, pois agora cada cópia terá sua própria ID e poderá ser diferenciada dentro do nosso sistema.

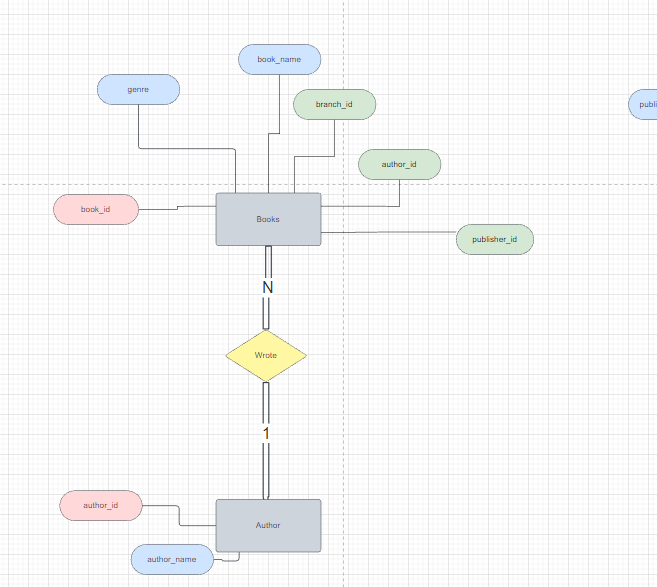
Vale a pena ressaltar que essa não é a única forma correta de estruturar essa nossa tabela. Se nós tivéssemos um atributo “Número De Cópias” que indica o número de cópias de cada livro que estão no sistema, então o nome do livro poderia passar a ser um atributo único, e poderia ser utilizado como nossa PK, com as cópias sendo diferenciadas através do atributo que criamos anteriormente, ou de uma nova tabela relacionada a ela. A razão de ter escolhido a identidade como PK é que ela facilita alguns processos, e faz mais sentido levando em conta outros aspectos do banco de dados, mas esse tipo de decisão é parte importante do processo de criação de um DB, e irá variar de caso em caso.

Agora que sabemos que o atributo em vermelho é a PK, podemos falar dos atributos em verde, que são conhecidos como Foreign Keys. Como você pode ter percebido, esses atributos são todos equivalentes a alguma outra table do nosso DB, e mais especificamente são equivalentes a PK dessas outras tables. Isso ocorre porque é utilizando essas FK que nós iremos criar as relações entre as tabelas do nosso DB, fazendo com que ele passe a ser um banco de dados relacional.

Existem 3 tipos principais de relações entre tables, e veremos cada um deles conforme eles forem aparecendo, mas essas relações vistas na table Livro são todas do mesmo tipo, a 1-to-N.





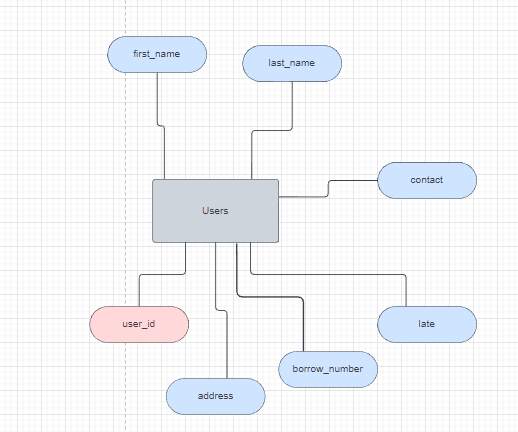


As 3 imagens acima, mostram a relação entre a table Livro, e as tables Editora, Filial e Autor, e mostram como podemos diagramar as relações para podermos estabelece-las corretamente na hora de programarmos o DB. No caso da relação 1-to-N, é caracterizada como uma relação na qual uma entidade A pode se relacionar com infinitas da entidade B, mas cada entidade B só pode se relacionar com uma única entidade A.

Por exemplo, uma editora pode publicar muitos livros, mas cada livro só é publicado por uma única editora, uma filial terá muitos livros, mas cada livro só pode estar presente em uma filial de cada vez, e um autor pode escrever muitos livros, mas cada livro só tem um autor.

Quando observamos uma relação desse tipo, para a representarmos corretamente no DB, basta adicionarmos uma FK a table que está no lado indicado como N, que nos 3 casos acima é a table Livros.

Usuário:



Esse é o diagrama da table de usuários, e dos atributos dela que queremos armazenar.

- user\_id: Identidade do usuário. Nossa PK para essa table, já que identifica unicamente cada usuário.

- first\_name: Primeiro nome do usuário.

- last\_name: Sobrenome do usuário.

- contact: Informação de contato do usuário. Pode ser um telefone, ou email, ou informação que nos permita contatar o usuário. Poderia também ser nossa PK se quiséssemos, mas é uma string, enquanto que normalmente é recomendado usar um número inteiro como PK.

- address: Endereço do Usuário. Se quiséssemos poderíamos separar esse atributo em sua própria tabela, em que cada coluna seria uma parte do endereço, para ter maior precisão, mas como essa não é uma coluna vital para o DB não considero necessário.

- borrow\_number: Número de Livros Emprestados. Como foi especificado no objetivo do DB, cada usuário só poderá estar com até 3 livros emprestados simultaneamente. Então precisamos armazenar quantos livros estão com cada usuário a cada momento.

- late: Atraso. Funciona como marcador de situação irregular do usuário. Se ele já estiver com 3 livros, ou estiver atrasado na devolução de um livro, o valor desse atributo será 1, e caso contrário será 0. Isso nos permite saber se um usuário tem permissão para pegar mais livros emprestados a qualquer momento.

Empréstimo:

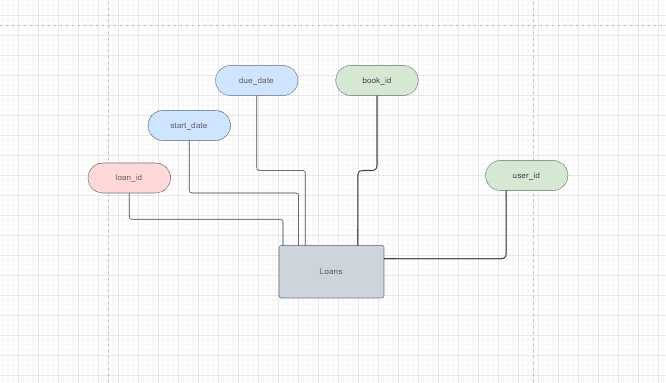


Diagrama da table Empréstimo, e seus atributos que queremos armazenar.

-loan\_id: Identidade do Empréstimo. PK dessa table, identifica unicamente cada um dos empréstimos.

-start\_date: Data inicial. A data na qual o livro foi emprestado.

-due\_date: Data de entrega. A data até a qual o livro deve ser devolvido.

-book\_id: Identidade do livro. Uma das FKs dessa table, indica qual livro foi emprestado.

-user\_id: Identidade do usuário. Uma das FKs dessa table, indical qual usuário está com esse livro.

Nessa table temos mais duas relações. A relação entre essa table e a table de Usuário é mais uma relação do tipo 1-to-N, que já vimos como deve ser realizada. A relação entre essa table e a table Livro entretanto, é do tipo 1-to-1.

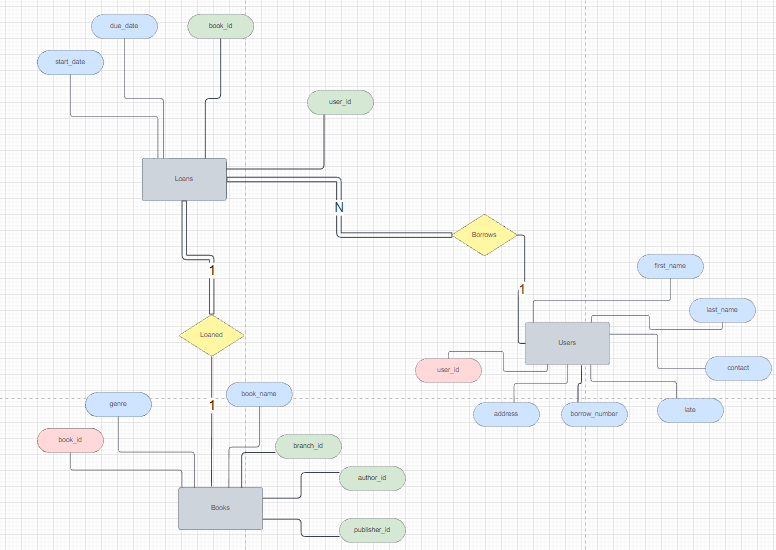


Diagrama da relação entre as tables Livro, Usuário e Empréstimo.

A forma que representamos essa relação é criando uma FK nessa table, similar ao caso 1-to-N, mas nesse caso devemos especificar que a FK é única, ou seja, todos os valores dessa FK devem ser diferentes. Essa relação é caracterizada, como uma relação na qual cada entidade A está relacionada a uma única entidade B, e cada entidade B está relacionada a uma única entidade A, como por exemplo, cada empréstimo só envolve um livro, e cada livro só pode estar envolvido em um único empréstimo.

Funcionário:

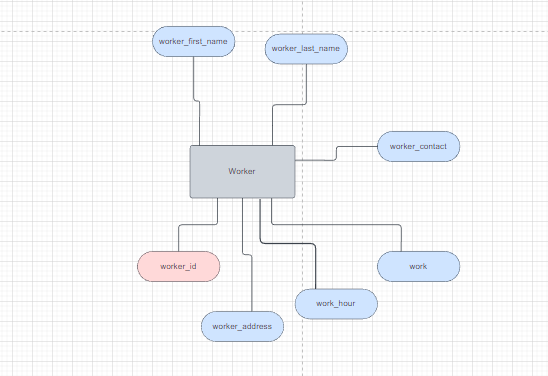


Diagrama da table Funcionário, e dos atributos que queremos armazenar.

-worker\_id – Identidade do funcionário. A PK dessa table, que identifica unicamente cada funcionário.

-worker\_first\_name – Primeiro nome do funcionário.

-worker\_last\_name – Sobrenome do funcionário.

-worker\_contact – Informação de contato do funcionário. Similar ao do usuário.

-work – Trabalho. Função exercida pelo funcionário.

-work\_hour – Carga horária do funcionário.

-worker\_address – Endereço do funcionário. Similar ao do usuário.

Filial:

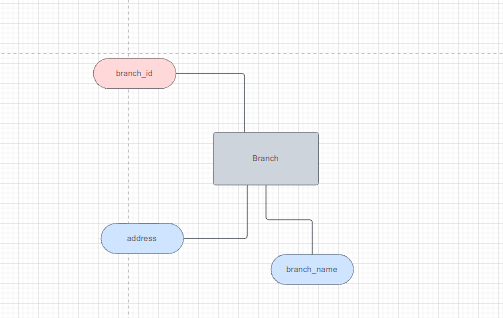


Diagrama da table Filial, e dos atributos que desejamos armazenar nela.

-branch\_id – Identidade da filial. PK da table, identifica unicamente cada uma das filiais.

-branch\_name – Nome da filial. Poderia ser usada como PK se tivéssemos certeza de que todas as filiais teriam nomes diferentes.

-address – Endereço da filial.

Olhando o diagrama não podemos perceber isso, mas essa tabela está relacionada a outra. Como vimos no objetivo do DB, cada filial tem múltiplos funcionários e cada funcionário pode trabalhar para mais de uma filial.

Isso significa uma relação N-to-M, caracterizada como uma relação na qual cada entidade A pode estar relacionada a múltiplas entidades B, e cada entidade B pode estar relacionada a múltiplas entidades A.

Essa relação é um pouco diferente das outras, pois para ser representada no DB não basta adicionar uma FK à uma das tables, é necessário criar uma nova tabela que irá armazenar a relação entre essas duas tables.

Emprego:

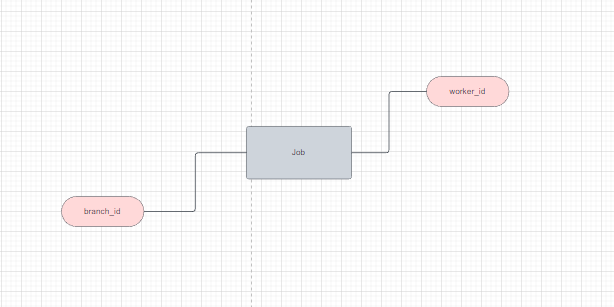
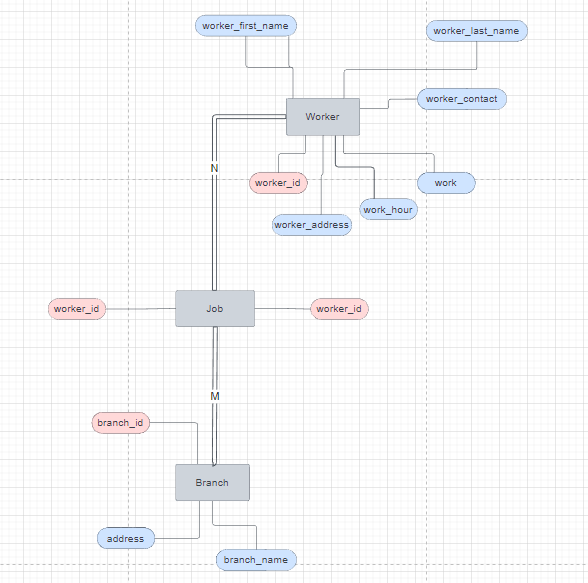


Diagrama da tabela Emprego, que representa a relação entre as tables Funcionário e Filial.

Esse é um tipo diferente de tabela, na qual a PK é composta por duas colunas diferentes ao mesmo tempo, ou seja, a combinação de “identidade do funcionário” e de “identidade da filial” deve ser única, mas podem ocorrer repetições nos valores de cada uma dessas colunas individualmente. Por exemplo, podemos ter o valor 1 na coluna A mais de uma vez desde que o valor da coluna B seja diferente em todos esses casos, e podemos ter o valor 1 na coluna B mais de uma vez desde que o valor da coluna A seja diferente em todos esses casos.

Esse tipo de tabela sempre deverá ser utilizado quando nos depararmos com uma relação do tipo N-to-M, e essas tabelas não precisam conter apenas as PKs correspondentes às tables relacionadas. Se quisermos podemos armazenar outras informações nessa table, desde que qualquer atributo que esteja incluído aqui seja dependente de ambas as PKs. Por exemplo, se quiséssemos adicionar a carga horária total de um funcionário, não poderíamos, pois a carga horária depende apenas da “identidade do funcionário”. Por outro lado, se quisermos adicionar a carga horária que o funcionário realiza em uma determinada filial, seria possível, pois as horas de trabalho que um funcionário realiza em uma filial depende tanto do funcionário quanto da filial na qual ele está trabalhando.

Podemos ver o diagrama da relação entre essas 3 tables na imagem a seguir:



Editora:

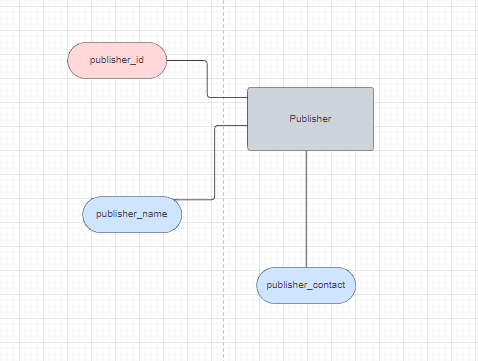


Diagrama da table Editora, e dos atributos que desejamos armazenar nela.

- publisher\_id – Identidade da editora. Essa é a PK dessa table, que identifica unicamente cada editora.

- publisher\_name – Nome da Editora. Poderia ser usado como PK já que toda editora tem obrigatoriamente um nome diferente, mas como mencionamos anteriormente, a PK idealmente deve ser um dígito.

- Publisher\_contact – Número de contato da editora. Importante de se armazenar já que a biblioteca negocia diretamente com as editoras para obter os livros.

Autor:

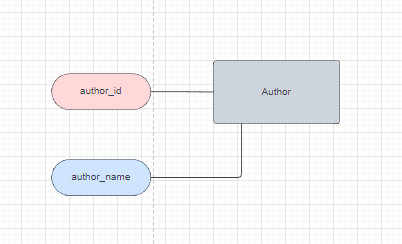


Diagrama da table Autor, e dos atributos que queremos armazenar nela.

- author\_id – Identidade do autor. PK que identifica unicamente cada autor.

- author\_name – Nome do Autor. Poderia ser utilizada como PK se tivéssemos garantia de que não existam 2 autores com o mesmo nome.

Essa tabela é a tabela mais simples do nosso banco de dados, mas ela é uma oportunidade interessante de elaborar sobre como nossas suposições sobre a relação entre tables afetam o nosso DB.

Como vimos anteriormente, se todos os autores tivessem um nome único, poderíamos usar o nome do autor como PK, e se fizéssemos isso essa tabela deixaria de ter uma razão para existir, já que seria uma tabela de uma coluna só. Isso significa que poderíamos eliminar essa tabela e simplesmente adicionar o nome do autor à table Livro, sem precisar utilizá-lo como FK, simplificando nosso DB.

Por outro lado, não é necessariamente verdade que cada livro tem apenas um autor. Livros técnicos, artigos científicos, HQs, ou até livros normais, muitas vezes tem mais do que um autor. Isso significa que se considerássemos essa possibilidade a relação entre Livro e Autor deixaria de ser uma relação 1-to-N, e passaria a ser N-to-M. Nesse caso não só nós precisaríamos de uma tabela para os autores, mas precisaríamos de uma tabela a mais para estabelecer a relação entre Livro e Autor.

Isso é uma demonstração importante de como a forma sobre a qual pensamos na relação entre cada elemento do banco de dados afeta como estruturamos o DB.

Todo esse planejamento pode parecer um pouco excessivo, mas bancos de dados relacionais não são simples de se alterar, cada tabela acaba dependendo de outras tabelas, e qualquer pequena mudança que você queira fazer frequentemente envolve alterar todas as tabelas relacionadas. Isso significa que se planejarmos com cuidado o DB inicialmente, reduziremos a chance de precisarmos refazer aspectos do banco de dados. Na seção Anexos você pode encontrar uma imagem que mostra um diagrama do banco de dados inteiro, de todas as tables e da relação entre elas.

Agora que determinamos todas as tables, determinamos os atributos de cada uma delas, e determinamos a relação entre elas, podemos finalmente começar a contruir de verdade nosso DB.

**3.3 - CREATE, DROP, SELECT e ALTER Tables**

A primeira coisa que devemos fazer após criar nosso banco de dados, é começar a criar as tables, como no exemplo a seguir.

CREATE TABLE book (

book\_id INT PRIMARY KEY AUTO\_INCREMENT,

    book\_name VARCHAR(100) NOT NULL,

    genre VARCHAR(30) NOT NULL,

    branch INT,

    author INT,

    publisher INT

    );

Na primeira linha nós iniciamos o comando CREATE TABLE, que irá criar a tabela, com o nome que especificarmos. Nesse caso estamos criando a tabela “book”, ou Livro, e depois de abrir o parêntese começamos a especificar os atributos presentes na tabela, que definirão as colunas da nosso table.

Começamos especificando nossa PK, primeiro escrevendo o nome que desejamos para ela, seguido do tipo de valor presente nessa coluna, e depois colocamos os modificadores que desejamos para essa coluna. Nesse caso declaramos que queremos que os valores da coluna sejam sempre números inteiros utilizando o termo INT, que essa coluna deve ser nossa PK utilizando o termo PRIMARY KEY, e que essa coluna deve ser incrementada automaticamente utilizando o termo AUTO\_INCREMENT.

A designação de PRIMARY KEY, indica apenas que todos os valores nessa coluna devem ser únicos, e não podem ser nulos. Poderíamos se quiséssemos, substituir o PRIMARY KEY pelos termos UNIQUE e NOT NULL, que teriam o mesmo efeito, mas veremos esses termos especificamente mais para frente.

O AUTO\_INCREMENT significa que toda vez que uma nova linha for inserida nessa tabela, o seu valor na coluna book\_id será automaticamente designado como o próximo valor disponível. Por exemplo, se o último livro nessa lista tem a book\_id 135, quando adicionarmos mais um livro à essa tabela, ele automaticamente terá a book\_id 136.

Seguindo para as próximas colunas, book\_name e genre, vemos dois novos comandos, o VARCHAR e NOT NULL. Como queremos armazenar nessas colunas o nome dos livros e o gênero literário deles, sabemos que os valores dessas colunas devem ser palavras. Quando quisermos armazenar caracteres, utilizamos o VARCHAR, de forma similar a qual utilizamos as strings em Python. O número entre parênteses indica o limite de caracteres para cada valor nessa coluna.

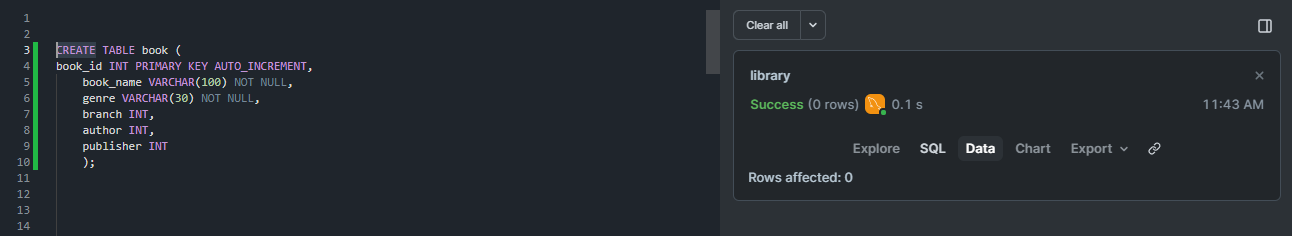
Além disso temos o NOT NULL, que indica que essas colunas precisam ter algum valor armazenado nelas, não podem ficar em branco. No geral é uma boa ideia sempre utilizar o NOT NULL, a não ser que exista alguma razão específica para que possam existir lacunas em branco nessa coluna.

Depois disso vamos para nossas FKs, branch, author e publisher. Quando estamos criando as FKs na table, o nome dessas colunas não precisa ser igual ao nome da coluna principal nas outras tabelas, a relação entre as tables só será estabelecida mais para frente, por enquanto estamos criando essas colunas apenas para guardar esse espaço na tabela.

A única coisa que precisamos garantir é que o tipo de valor nessas colunas seja igual ao tipo de valor das colunas principais das tabelas que queremos relacionar. Como as PKs de branch, author, e publisher, são todas números inteiros, utilizamos INT nessa tabela.

Em relação a sintaxe, fique atento as virgulas que separam cada linha dentro dos parênteses, ao início e fim dos parênteses, e sempre termine qualquer comando com o ponto e vírgula.

Agora que vimos como criar uma table, podemos executar esse query, que deve funcionar corretamente. A sua tela deve estar assim:



Isso significa que nossa table foi criada com sucesso. Preste atenção na coluna verde perto dos números das linhas do código, ela indica qual parte do seu programa está será executada quando apertarmos o botão “Run”.

Agora vamos ver nossa table e para isso vamos usar o comando SELECT.

SELECT \* FROM book;

O comando SELECT nos deixa ler a informação presente nas tables. Depois do SELECT nós especificamos quais colunas da table nós queremos selecionar, que no caso de um \*, significa todas as colunas. Depois escolhemos a table que queremos selecionar utilizando FROM seguido do nome da table desejada. Então podemos ler esse comando como “Selecione todas as colunas da table book”.

Se executarmos essa linha, veremos então que isso é exatamente o que ela faz. As colunas estarão vazias, já que ainda não inserimos nenhuma linha nessa tabela, mas já podemos ver os nomes das colunas que determinamos durante a criação da tabela.

Agora que aprendemos a criar uma table, vamos aprender a deletar uma table. Para fazer isso simplesmente utilizamos o comando DROP TABLE.

DROP TABLE book;

Execute essa linha, e depois tente ler a tabela como fizemos anteriormente para ver que ocorrerá um erro, no qual a nossa table book não existe mais. Essa é uma boa demonstração de como é fácil fazer modificações enormes, e possivelmente desastrosas, no nosso banco de dados. Felizmente como acabamos de iniciar nosso DB, basta executarmos o comando CREATE TABLE que fizemos anteriormente. Vale ressaltar que, no PopSQL você não precisa deletar o código que escrevemos anteriormente, já que podemos executar o código por partes.

Agora vamos criar o resto das tables:

CREATE TABLE author ( #Tabela Autor

author\_id INT PRIMARY KEY AUTO\_INCREMENT,

    author\_name varchar(40) NOT NULL

);

CREATE TABLE user ( #Tabela Usuário

user\_id INT PRIMARY KEY AUTO\_INCREMENT,

    first\_name VARCHAR(15) NOT NULL,

    last\_name VARCHAR(15) NOT NULL,

    contact VARCHAR(30) NOT NULL,

    borrow\_number INT NOT NULL,

    address VARCHAR(30) NOT NULL,

    late BOOLEAN NOT NULL

);

CREATE TABLE publisher ( #Tabela Editora

publisher\_id INT PRIMARY KEY AUTO\_INCREMENT,

    publisher\_name VARCHAR(40) NOT NULL,

    publisher\_contact VARCHAR(40) NOT NULL

);

CREATE TABLE branch ( #Tabela Filial

branch\_id INT PRIMARY KEY AUTO\_INCREMENT,

    address VARCHAR(40) NOT NULL,

    branch\_name VARCHAR(40) NOT NULL

);

CREATE TABLE loan ( #Tabela Empréstimo

loan\_id INT PRIMARY KEY AUTO\_INCREMENT,

    start\_date DATE NOT NULL,

    due\_date DATE NOT NULL,

    book INT UNIQUE,

    user INT

);

CREATE TABLE worker ( #Tabela Funcionário

worker\_id INT PRIMARY KEY AUTO\_INCREMENT,

    worker\_first\_name VARCHAR(40) NOT NULL,

    worker\_last\_name VARCHAR(40) NOT NULL,

    worker\_contact VARCHAR(40) NOT NULL,

    work VARCHAR(40) NOT NULL,

    work\_hour INT NOT NULL,

    worker\_address VARCHAR(40) NOT NULL

);

Esses são os comandos para criar as outras tabelas, e todos usam exatamente a mesma lógica que eu demonstrei na criação da table book. As únicas diferenças são o uso do símbolo #, o uso de um novo tipo de valor nas colunas, e o uso de Unique.

O símbolo # é utilizado quando queremos criar comentários em nosso código, isto é, o que está imediatamente depois de # não será considerado como parte do código, e está lá apenas para o benefício de simplificar a leitura.

Na table user, ou Usuário, vemos a coluna late, que tem como tipo de valor BOOLEAN, que indica que os únicos valores que podem estar presentes nela são True e False.

Na tabela loan, ou Empréstimo, quando declaramos a coluna book, que será eventualmente transformada em uma FK, utilizamos o termo UNIQUE para indicar que todos os valores nessa coluna devem ser únicos. Isso é importante para manter a relação 1-to-1 entre a tabela book e a tabela loan que estabelecemos durante a fase de planejamento do DB.

Agora falta só criarmos a tabela job, ou Emprego, mas ela é um pouco diferente das outras.

CREATE TABLE job (

    worker\_id INT,

    branch\_id INT,

    PRIMARY KEY(worker\_id, branch\_id),

    FOREIGN KEY(worker\_id) REFERENCES worker(worker\_id) ON DELETE CASCADE,

    FOREIGN KEY(branch\_id) REFERENCES branch(branch\_id) ON DELETE CASCADE

);

Como vimos anteriormente essa tabela existe exclusivamente para estabelecer a relação entre as tabelas worker e branch, então precisamos estabelecer essa relação ao criarmos ela. Nas duas primeiras linhas estabelecemos as colunas worker\_id e branch\_id como contendo apenas números inteiros.

Depois disso declaramos nossa PK, composta pela combinação das duas colunas que definimos anteriormente. Essas colunas além de juntas formarem a PK, são individualmente FKs das tabelas que estão relacionando.

Vamos ver esses comandos de um em um. Primeiro o comando Foreign Key determina que a coluna entre os parênteses é uma FK. Depois REFERENCES determina qual coluna nossa FK está referenciando, nesse caso a coluna worker\_id que está presente na table worker. Esse comando então pode ser traduzido como “worker\_id é uma FK, que faz referência a coluna worker\_id que está presente na tabela worker”. Fazemos então a mesma coisa só que com a coluna branch\_id.

A última parte desses comandos é o ON DELETE CASCADE. O comando ON DELETE é utilizado em FKs, para determinar o que deve acontecer com uma FK caso o valor associado à ela seja deletado da table original, e existem dois tipos principais de ON DELETE, o SET NULL e o CASCADE. No SET NULL, ao se deletar uma informação da tabela original, os valores da outra tabela que dependem dessa informação são substituídos pelo valor NULL. No caso do CASCADE, ao deletarmos o valor da tabela original, iremos deletar também todas as linhas da outra tabela que dependiam desse valor.

Por exemplo, digamos que o funcionário com a worker\_id = 1 trabalhe em duas filiais, que tem as branch\_id = 1 e branch\_id = 2, respectivamente. Isso resultará em duas linhas nessa tabela, uma linha (1,1) e uma linha (1,2). Agora, imagine que a filial 2 foi fechada e, portanto, deletamos ela da nossa tabela de filiais. Caso o delete seja SET NULL, nossa tabela passará a ter duas linhas, uma (1,1) e uma (1, Null). Caso o delete seja CASCADE, passaremos a ter apenas uma linha, com valores (1,1).

No geral, só devemos utilizar CASCADE em situações em que temos certeza de que a tabela com a FK depende completamente da tabela original. Por exemplo se um funcionário é demitido, e logicamente, para de trabalhar para a biblioteca, ele será removido da tabela de funcionários. Portanto, ele será obviamente removido também da tabela Emprego,

Com isso, todas as nossas tabelas já foram criadas, mas as relações entre elas ainda não foram totalmente estabelecidas. Para isso, precisaremos utilizar o comando ALTER TABLE.

ALTER TABLE book

ADD FOREIGN KEY(branch)

REFERENCES branch(branch\_id)

ON DELETE SET NULL;

O comando ALTER TABLE é utilizado sempre que queremos mudar, de alguma forma, a estrutura da tabela. Se quisermos mudar nossa PK, decidirmos permitir valores NULL em uma coluna, alterarmos o tipo de valor da coluna de VARCHAR para INT, ou como nesse caso, transformar uma coluna da tabela em uma FK, precisaremos sempre utilizar o ALTER TABLE.

Na primeira linha designamos a tabela que queremos alterar, nesse caso a tabela book. Depois usamos ADD FOREIGN KEY, para transformar a coluna que colocarmos entre parênteses, nesse caso a coluna branch da tabela book, em uma FK.

Então nós especificamos qual coluna, de qual table, é utilizada como referencia para a nossa FK, que nesse caso é a coluna branch\_id da table branch. Por fim, especificamos o que deve acontecer caso o valor de branch\_id seja deletado, usando ON DELETE SET NULL.

Precisamos agora fazer a mesma coisa para criar as outras relações que estabelecemos na fase de planejamento do nosso DB.

ALTER TABLE book #Estabelece FK Autor na tabela Livro

ADD FOREIGN KEY(author)

REFERENCES author(author\_id)

ON DELETE SET NULL;

ALTER TABLE book #Estabelece FK Editor na tabela Livro

ADD FOREIGN KEY(publisher)

REFERENCES publisher(publisher\_id)

ON DELETE SET NULL;

ALTER TABLE loan #Estabelece FK Usuário na tabela Empréstimo

ADD FOREIGN KEY(user)

REFERENCES user(user\_id)

ON DELETE SET NULL;

ALTER TABLE loan #Estabelece FK Livro na tabela Empréstimo

ADD FOREIGN KEY(book)

REFERENCES book(book\_id)

ON DELETE SET NULL;

Lembrando que a FK de uma tabela A, deve sempre referenciar uma PK de uma tabela B, para estabelecer a relação entre elas. Podemos agora usar o comando SELECT para ver se todas as tabelas estão montadas corretamente.

Agora que vimos como criar, deletar, estruturar e ler nossas tabelas, podemos começar a inserir valores nelas.

**3.3 – Insert, Update e Delete Tables**

Os comandos que vimos anteriormente são utilizados para alterar as estruturas das tabelas, e agora veremos os comandos que afetam os valores armazenados nas tabelas.

A melhor forma de fazer isso é preenchendo as tabelas, e para isso precisamos utilizar o comando INSERT INTO.

INSERT INTO user VALUES

(1, 'Haleema', 'Wallis', '(937) 211-3813', 1, 'A street', FALSE );

Utilizamos o comando INSERT INTO quando queremos introduzir uma linha em uma tabela, e para isso precisamos primeiro especificar a tabela na qual queremos introduzir os valores, que nesse caso é a tabela user. Depois precisamos usar VALUES para especificar que valores queremos inserir na tabela.

Dentro dos parênteses, escrevemos os valores desejados de um em um, separados por vírgula, e atentos ao fato de que precisamos introduzir um número de valores igual ao número de colunas da tabela, que nesse caso é 7. Precisamos também escrever esses valores na mesma ordem que definimos as colunas durante a criação das tabelas.

Nessa tabela temos as colunas user\_id, first\_name, last\_name, contact, borrow\_number, address e late, logo precisamos inserir os valores nessa ordem. Primeiro a user\_id que será 1 já que é o primeiro usuário introduzido, depois o primeiro nome do usuário, depois um sobrenome do usuário, a informação de contato do usuário, o número de livros que esse usuário pegou emprestado, o endereço do usuário e por fim um valor True ou False. Nesse caso, esses dados foram gerados aleatoriamente, mas no mundo real seriam fornecidos pelo próprio usuário durante o cadastro.

Esse é só um usuário, mas se quisermos podemos introduzir vários de uma vez só da seguinte forma:

INSERT INTO user VALUES

(2, 'Miya', 'Day', '(575) 520-9050', 0, 'B street', FALSE ),

(3, 'Kelsie', 'Pierce', '(635) 884-8071', 2, 'C street', FALSE ),

(4, 'Nayla', 'Kendall', '(886) 798-6941', 1, 'D street', FALSE ),

(5, 'Alejandro', 'Bassett', '(460) 329-0224', 0, 'Avenue A', FALSE ),

(6, 'Alyssia', 'Hewitt', '(557) 613-0263', 0, 'Avenue B', FALSE ),

(7, 'Malik', 'Bouvet', '(664) 219-7553', 3, 'A boulevard', FALSE ),

(8, 'Yasemin', 'Mcneil', '(248) 261-1707', 0, 'F street', FALSE ),

(9, 'Kiki', 'Erickson', '(520) 949-8517', 1, 'R street', TRUE ),

(10, 'Chanel', 'Walton', '(496) 999-3850', 0, 'G street', FALSE ),

(11, 'Edwin', 'Brandt', '(859) 468-8894', 2, 'Avenue Q', FALSE ),

(12, 'Bo', 'Benitez', '(611) 791-4583', 2, 'Avenue Z', FALSE ),

(13, 'Franky', 'Rigby', '(438) 459-4171', 0, 'P boulevard', FALSE ),

(14, 'Ivo', 'Atherton', '(251) 814-3375', 3, 'A street', TRUE ),

(15, 'Ian', 'Taylor', '(522) 756-6527', 0, 'J street', FALSE )

;

Isso nos dará uma lista razoavelmente completa de usuários que podemos usar para testar nosso DB. Use o comando SELECT que vimos anteriormente para ver que agora, a tabela estará preenchida por esses valores.

Agora podemos fazer isso com todas as outras tabelas, levando em conta que devemos começar primeiro pelas tabelas que não contém Foreign Keys, e por último pelas tabelas que contém Foreign Keys, porque se introduzirmos um valor que depende de outra tabela, e essa outra tabela estiver vazia, resultará em um erro.

INSERT INTO worker VALUES

(1,"Megan","Hawkins","(748) 297-6731",'janitor',40,"A street"),

(2,"Luke","Barlow","(552) 607-2153",'janitor',40,"B street"),

(3,"Vinnie","Haines","(550) 336-1739",'janitor',40,"D street"),

(4,"Ezekiel","Dean","(764) 263-3028",'janitor',40,"C street"),

(5,"Elizabeth","Cox","(528) 886-7203",'janitor',20,"Q street"),

(6,"Adem","Aguirre","(477) 578-2097",'janitor',20,"G street"),

(7,"Sian","Sparks","(523) 974-7431",'librarian',40,"O street"),

(8,"Malachi","Jensen","(672) 361-0124",'librarian',40,"Avenue C"),

(9,"Adele","Bishop","(496) 814-8453",'librarian',40,"Avenue J"),

(10,"Kyron","Stark","(662) 911-5488",'librarian',40,"V street"),

(11,"Cory","Lloyd","(254) 628-5884",'librarian',45,"P street"),

(12,"Elmer","Foster","(409) 783-1161",'librarian',45,"I street"),

(13,"Gideon","White","(644) 254-2898",'manager',50,"D street")

;

INSERT INTO branch VALUES

(1, 'Street A', 'Branch A'),

(2, 'Street B', 'Branch B'),

(3, 'Street C', 'Branch C'),

(4, 'Avenue Q', 'Branch Q');

INSERT INTO job VALUES

(1,1),

(2,2),

(3,3),

(4,4),

(5,1),

(6,2),

(7,1),

(8,2),

(9,3),

(10,4),

(11,1),

(11,2),

(12,3),

(12,4),

(13,1),

(13,2),

(13,3),

(13,4)

;

INSERT INTO author VALUES

(1, 'Stephen King'),

(2, 'J. K. Rowling'),

(3, 'George Orwell'),

(4, 'Agatha Christie'),

(5, 'J. R. R. Tolkien'),

(6, 'Arthur Conan Doyle'),

(7, 'C. S. Lewis'),

(8, 'Haruki Murakami'),

(9, 'Ernest Hemingway'),

(10, 'Mark Twain'),

(11, 'Jane Austen'),

(12, 'Isaac Asimov'),

(13, 'T. S. Eliot'),

(14, 'Alexandre Dumas'),

(15, 'Robert A. Heinlein'),

(16, 'Ursula K. Le Guin'),

(17, 'Truman Capote'),

(18, 'Hunter S. Thompson'),

(19, 'Leo Tolstoy'),

(20, 'Alan Moore'),

(21, 'Neil Gaiman');

INSERT INTO publisher VALUES

(1, 'Publisher A' , '(398) 679-2916'),

(2, 'Publisher B' , '(401) 372-3311'),

(3, 'Publisher C' , '(274) 706-4702'),

(4, 'Publisher D' , '(513) 271-2096'),

(5, 'Publisher E' , '(518) 755-1915'),

(6, 'Publisher F' , '(270) 394-2790');

INSERT INTO book VALUES

(1, 'IT', 'Horror',1,1,1),

(2, 'IT', 'Horror',2,1,3),

(3, 'Green Mile', 'Magic Realism',3,1,3),

(4, "Harry Potter and the Philosopher's Stone", 'Fantasy',4,2,4),

(5, 'Harry Potter and the Goblet of Fire', 'Fantasy',4,2,4),

(6, '1984', 'Dystopian',1,3,5),

(7, 'Animal Farm', 'Political Satire',1,3,5),

(8, 'And Then There Were None', 'Mystery',2,4,6),

(9, 'And Then There Were None', 'Mystery',2,4,6),

(10, 'Lord of the Rings', 'Fantasy',1,5,2),

(11, 'Lord of the Rings', 'Fantasy',2,5,2),

(12, 'Lord of the Rings', 'Fantasy',3,5,2),

(13, 'Lord of the Rings', 'Fantasy',4,5,2),

(14, 'Sherlock Holmes: The Complete Novels and Stories Volume II', 'Mystery',4,6,1),

(15, 'Sherlock Holmes: The Complete Novels and Stories Volume I', 'Mystery',4,6,1),

(16, 'Prince Caspian', 'Fantasy',3,7,2),

(17, 'The Lion, the Witch and the Wardrobe', 'Fantasy',3,7,2),

(18, 'Norwegian Wood', 'Romance',3,8,1),

(19, 'The Wind-Up Bird Chronicle', 'Fantastique',3,8,5),

(20, 'For Whom the Bell Tolls', 'War',1,9,4),

(21, 'The Old Man and the Sea', 'Literary Fiction',1,9,4),

(22, 'The Adventures of Tom Sawyer', 'Folk',2,10,1),

(23, 'Adventures of Huckleberry Finn', 'Folk',2,10,1),

(24, 'Sense and Sensibility', 'Romance',2,11,6),

(25, 'Pride and Prejudice', 'Romance',2,11,6),

(26, 'Foundation', 'Sci-fi',1,12,5),

(27, 'Foundation and Empire', 'Sci-fi',1,12,5),

(28, 'Second Foundation', 'Sci-fi',4,12,5),

(29, 'The Waste Land', 'Poetry',1,13,3),

(30, 'Old Possums Book of Practical Cats', 'Poetry',1,13,3),

(31, 'The Count of Monte Cristo', 'Adventure',1,14,2),

(32, 'The Count of Monte Cristo', 'Adventure',4,14,2),

(33, 'The Three Musketeers', 'Adventure',1,14,2),

(34, 'Starship Troopers', 'Sci-fi',4,15,5),

(35, 'Stranger in a Strange Land', 'Sci-fi',3,15,5),

(36, 'The Moon Is a Harsh Mistress', 'Sci-fi',2,15,5),

(37, 'A Wizard of Earthsea', 'Fantasy',2,16,6),

(38, 'The Tombs of Atuan', 'Fantasy',2,16,6),

(39, 'The Left Hand of Darkness', 'Sci-fi',4,16,3),

(40, 'In Cold Blood', 'True Crime',4,17,4),

(41, "Hell's Angels", 'Journalism',4,18,4),

(42, 'Fear and Loathing in Las Vegas', 'Journalism',4,18,4),

(43, 'War and Peace', 'Literary Fiction',2,19,2),

(44, 'Anna Karenina', 'Literary Fiction',2,19,2),

(45, 'The Kingdom of God Is Within You', 'Theology',3,19,2),

(46, 'Watchmen', 'Comics',1,20,2),

(47, 'V for Vendetta', 'Comics',1,20,2),

(48, 'Sandman', 'Comics',1,21,2),

(49, 'American Gods', 'Fantasy',1,21,2);

INSERT INTO loan VALUES

(1, "2022-10-25","2022-11-25", 1, 1),

(2, "2022-10-13","2022-11-13", 2, 2),

(3, "2022-10-14","2022-11-14", 3, 3),

(4, "2022-10-15","2022-11-15", 4, 4),

(5, "2022-10-25","2022-11-25", 5, 5),

(6, "2022-10-03","2022-11-03", 6, 6),

(7, "2022-10-28","2022-11-28", 7, 7),

(8, "2022-10-23","2022-11-23", 8, 8),

(9, "2022-10-11","2022-11-11", 9, 9),

(10, "2022-10-12","2022-11-12", 10, 10),

(11, "2022-10-30","2022-11-30", 11, 11),

(12, "2022-10-24","2022-11-24", 12, 12),

(13, "2022-10-07","2022-11-07", 13, 13),

(14, "2022-10-29","2022-11-29", 14, 14),

(15, "2022-10-11","2022-11-11", 15, 15);

Usando o comando SELECT, podemos ver se todas as informações que introduzimos foram inseridas corretamente nas tabelas.

Agora que vimos como inserir linhas nas nossas tabelas, vamos ver como modificar os valores que estão dentro das tabelas. Para isso usaremos o comando UPDATE, como podemos ver nesse trecho.

UPDATE user

SET borrow\_number = 1

WHERE user\_id = 2;

A maioria dos comandos UPDATE vão ter uma estrutura bem similar a essa, então vamos analisar esse código de linha em linha. Na primeira linha usamos o comando UPDATE para definir qual tabela queremos modificar, que nesse caso é a tabela user. Depois, podemos escrever o que queremos mudar dentro dessa tabela.

Existe uma grande variedade de comandos que podemos usar nessa parte, e veremos muitos deles eventualmente, mas nesse caso utilizamos SET, que simplesmente faz com que o valor passe a ser igual ao que quisermos. Nesse caso estou fazendo com que o borrow\_number, o número de livros que o usuário retirou, passe a ser igual a 1.

Por fim, especificamos o local no qual queremos realizar essa mudança, utilizando o comando WHERE. Nesse caso, essa mudança será feita na linha na qual a user\_id for igual a 2, ou seja, agora a tabela mostrará que o usuário 2 está com 1 livro retirado.

Vale a pena ressaltar que é muito importante que utilizemos o comando WHERE, pois caso ele não fosse usado nesse exemplo, o valor da coluna borrow\_number passaria a ser 1 para TODOS os usuários.

Já vimos como inserir valores, e como modificar valores, então vejamos como deletar valores, utilizando o comando DELETE.

O comando DELETE é diferente do DROP, pois o DROP remove a tabela em si, enquanto o DELETE remove os valores armazenados na tabela. A sua estrutura é similar a do UPDATE, mas ainda mais simples.

Digamos que um usuário tenha acabado de devolver todos os livros que tinha retirado, e ao invés de queremos então retirar esses livros da lista de empréstimos, basta executarmos o seguinte comando.

DELETE FROM loan

WHERE user = 5;

Na primeira linha determinamos que queremos deletar valores da tabela loan, e na segunda linha especificamos que queremos deletar as linhas nas quais o usuário seja igual a 5, que seria o usuário do nosso exemplo. Como no caso do update, se não utilizarmos o WHERE para especificar as linhas desejadas, teríamos deletado todos os valores da tabela.

**3.4 - Queries**

Anteriormente vimos como utilizar o SELECT para ler as informações de uma tabela de uma forma básica, mas agora veremos a verdadeira profundidade e especificidade com a qual podemos analisar dados utilizando SQL.

SELECT book\_id, book\_name, genre, author FROM book;

Antes, estávamos selecionando as tabelas, inteiras, mas podemos também selecionar apenas algumas colunas específicas. Nesse caso, ao executarmos esse comando receberemos uma tabela com apenas as 4 colunas que especificamos.

Porém, essa tabela não é tão útil, já que na coluna do autor nós só vemos números ao invés de nomes. Para isso precisamos conectar a tabela book e a tabela author, utilizando as FKs que estabelecemos anteriormente.

SELECT book.book\_id, book.book\_name, book.genre, author.author\_name FROM book

JOIN author

ON book.author = author.author\_id;

Na primeira linha estamos selecionando as mesmas colunas de antes, mas estamos especificando a qual tabela elas pertencem. Por exemplo, “book.book\_id” significa “coluna book\_id da tabela book”.

Essa especificidade não é necessariamente obrigatória, já que só ocorrerá algum erro se tiver uma coluna com o mesmo nome nas duas tabelas, mas é uma boa ideia sempre utilizar esse método para facilitar a leitura.

Na segunda linha nós especificamos a tabela com a qual queremos combinar a nossa tabela inicial, então nesse caso, queremos juntar a tabela author à tabela book. Na terceira linha especificamos em qual coluna vamos realizar a junção das tabelas.

Como estabelecemos anteriormente, a coluna author da tabela book contém a mesma informação que a coluna author\_id da tabela author, então é nessa coluna que realizaremos a junção. Isso nos retornará a mesma tabela do exemplo anterior, mas dessa vez, teremos o nome do autor ao invés do número do autor.

Se quisermos, saber em qual filial cada livro está, e qual editora publica esse livro basta juntarmos mais tabelas de uma vez.

SELECT book\_name, genre, author\_name, branch.branch\_name, branch.address, publisher.publisher\_name FROM book

JOIN author

ON book.author = author.author\_id

JOIN branch

ON book.branch = branch.branch\_id

JOIN publisher

ON book.publisher = publisher.publisher\_id;

Nesse exemplo nós retiramos a coluna book\_id, e adicionamos as colunas branch\_name, branch.address, e publisher\_name. Para poder obter esses valores simplesmente realizamos o processo de junção mais duas vezes, seguindo exatamente a mesma lógica, mas dessa vez com as tabelas branch e publisher, conectadas à tabela book.

Essa é uma boa lista, com muita informação, mas está desorganizada. Digamos que estejamos interessados em organizar essa lista por gênero literário, podemos então usar um comando que realiza isso para nós.

SELECT book\_name, genre, author\_name, branch.branch\_name, branch.address, publisher.publisher\_name FROM book

JOIN author

ON book.author = author.author\_id

JOIN branch

ON book.branch = branch.branch\_id

JOIN publisher

ON book.publisher = publisher.publisher\_id

ORDER BY book.genre DESC;

Usando o comando ORDER BY, nós podemos ordenar as listas da forma que quisermos. Nesse caso, a lista está ordenada por gênero, em ordem descendente, como é indicado pelo DESC. Ou seja, os livros do mesmo gênero estão todos agrupados na lista, e esses grupos em si estão ordenados por ordem alfabética inversa.

Mas digamos que a maioria das filiais sejam muito distantes de nós, e só possamos na prática retirar os livros nas filiais “Branch A” e “Branch C”. Podemos então usar mais um comando para filtrar apenas pelas filiais que estão próximas de nós.

SELECT book\_name, genre, author\_name, branch.branch\_name, branch.address, publisher.publisher\_name FROM book

JOIN author

ON book.author = author.author\_id

JOIN branch

ON book.branch = branch.branch\_id

JOIN publisher

ON book.publisher = publisher.publisher\_id

WHERE branch.branch\_name = "Branch A" OR branch.branch\_name = "Branch C"

ORDER BY book.genre DESC;

Utilizamos o comando WHERE para especificar que queremos apenas as linhas nas quais o nome da filial seja igual a “Branch A” OU igual a “Branch B”. Como podemos ver utilizando os queries podemos obter qualquer informação que quisermos, independente de quão específica ela seja.

A maioria dos queries executados até agora utilizaram o JOIN, porém esse é só um dos tipos de junção que podemos utilizar. Existe também o LEFT JOIN, o RIGHT JOIN e o FULL JOIN.

Primeiro, execute os comandos a seguir:

CREATE TABLE popularity (

genre\_name VARCHAR(40) PRIMARY KEY,

popularity\_percent INT NOT NULL);

INSERT INTO popularity VALUES

("Fantasy", 30),

("Sci-fi", 25),

("Romance", 10),

("Comics", 10),

("Mystery", 15),

("Non-Fiction", 10);

Isso vai criar uma nova tabela chamada popularidade, que nos dará o nome de um gênero e qual percentual de nossos usuários considera esse seu tipo favorito de livro. Podemos então comparar com nossos livros, para determinar qual tipo de livros devemos priorizar. Executando um JOIN normal temos:

SELECT DISTINCT book.genre, popularity.popularity\_percent FROM book

JOIN popularity

ON book.genre = popularity.genre\_name;

Esse commando DISTINCT junto ao SELECT especifica que podemos ignorar linhas duplicadas. Se executarmos isso, receberemos uma lista de gêneros, com suas popularidades ao lado, porém nem todos os gêneros da tabela popularity estarão visíveis.

Isso ocorre pois o JOIN comum considera apenas a interseção das duas tabelas, e um dos gêneros considerados populares não está presente na nossa lista de livros e ,portanto, não faz parte da interseção.

Para consertarmos isso basta utilizarmos o LEFT JOIN:

SELECT DISTINCT book.genre, popularity.popularity\_percent FROM book

LEFT JOIN popularity

ON book.genre = popularity.genre\_name;

Essa lista mostrará todos os gêneros que não estão na lista de popularidade, e irá colocar NULL como valor na coluna de popularidade. O RIGHT JOIN é similar, mas nesse caso veremos todos os valores de popularidade, enquanto que o nome do gênero que não está presente na lista será substituído pelo valor NULL.

SELECT DISTINCT book.genre, popularity.popularity\_percent FROM book

RIGHT JOIN popularity

ON book.genre = popularity.genre\_name;

Por último temos o FULL JOIN, que é realizado ao se combinar um LEFT e RIGHT JOINS. Ele mostrará todos os valores do gênero e todos os valores de popularidade na mesma tabela.

SELECT DISTINCT  book.genre, popularity.popularity\_percent FROM book

LEFT JOIN popularity

ON book.genre = popularity.genre\_name

UNION

SELECT DISTINCT  book.genre, popularity.popularity\_percent FROM book

RIGHT JOIN popularity

ON book.genre = popularity.genre\_name;

**3.5 – CREATE e CALL Procedures**

Agora que estamos mais familiarizados com queries, podemos começar a criar o que chamamos PROCEDURES, ou procedimentos. Esses procedimentos são bem parecidos com as funções que criamos em python anteriormente, e tem o mesmo objetivo, criar uma forma fácil de a qualquer momento executar um código mais complexo. Nessa seção vamos ver como criar esses procedimentos, e veremos todos os procedimentos que usaremos durante a integração com python.

A primeira coisa que temos que fazer é sair do PopSQL e entrar no MySQL Workbench que instalamos anteriormente, cuja razão será explicada em breve. Ao abrir o workbench, devemos ver uma lista de conexões, onde poderemos ver nosso server MySQL. Clique nele, e use a senha para entrar.

Depois disso devemos chegar em uma tela similar a do PopSQL, mas antes de começar a criar procedimentos precisamos garantir que estamos conectados ao nosso DB. Clique na aba database, e selecione “connect to database”. No espaço “Stored Connection”, selecione “Local Instance MySQL80”, e no espaço “default schema” escreva o nome do nosso DB e aperte OK. Com isso a conexão deve ter sido estabelecida. Para confirmar, tente acessar uma de nossas tabelas.

Um problema do nosso DB até agora é que a relação entre certos valores ainda não está estabelecida. Por exemplo, nossa coluna borrow\_number, armazena o número de livros que cada usuário pegou emprestado simultaneamente. Entretanto, não existe nenhum mecanismo no nosso DB que garanta que esse número esteja correto, afinal, ao adicionarmos uma entrada na tabela empréstimo, não ocorre modificação na coluna borrow\_number.

Então nosso primeiro procedimento irá checar quantos livros cada usuário tem emprestado no momento, e armazenar esse número corretamente na coluna borrow\_number. O código que vemos a seguir faz exatamente isso, e iremos analisa-lo passo-a-passo.

delimiter $$

CREATE PROCEDURE check\_borrow()

BEGIN

    SET @y1 = 1;

SET @max\_num\_1= (SELECT count(user\_id) FROM user);

    loop\_label: LOOP

        IF @y1 > @max\_num\_1 THEN

            LEAVE loop\_label;

        END IF;

        SET @countage = (SELECT COUNT(user) from loan where user = @y1);

        UPDATE user

        SET borrow\_number = @countage

        where user\_id = @y1;

        SET @y1 = @y1 + 1;

        ITERATE loop\_label;

    END LOOP loop\_label;

END$$

delimiter ;

A primeira coisa que percebemos é o termo Delimiter, que é a razão de utilizarmos o workbench. O que esse termo faz é alterar o caractere que representa o fim de um query. Normalmente o ponto e vírgula indicam o fim de um query, mas ao utilizar o delimiter estamos mudando o caractere do ponto e vírgula para, nesse caso, $$.

Isso é importante porque dentro de um procedimento geralmente existem vários queries ao mesmo tempo, precisamos usar o ponto e vírgula para diferenciar esses queries internos, porém, se não usássemos o delimiter todo ponto e vírgula encerraria nossa função. Pense no delimiter como o equivalente das aspas triplas que vimos anteriormente no python, utilizando-as nós podemos armazenar aspas normais nas nossas strings sem encerrar a string. Repare que a última linha desse procedimento é outro delimiter, que faz com que o ponto e vírgula volte a ser o indicador de fim do query.

Depois do delimiter temos o comando CREATE PROCEDURE, que definie o nome do procedimento que iremos criar, chamado nesse caso de check\_borrow(). O comando BEGIN indica o início do procedimento.

Nossas duas linhas seguintes estão declarando variáveis, que no SQL tem sempre o símbolo @ antes dos seus nomes. A variável y1 se inicia com o valor de 1, e ela será utilizada ao longo do procedimento para indicar o número do usuário que estamos checando naquele momento.

A variável max\_num\_1, tem um valor igual a um query, que está especificado entre parênteses. O comando COUNT faz a contagem do número de valores associados a uma determinada coluna de uma determinada tabela, nesse caso a coluna user\_id da tabela user. Isso na prática significa que essa função conta o número de usuários cadastrados no sistema, e essa informação é então armazenada na variável max\_num\_1.

Isso nos leva ao comando LOOP. Essa linha está declarando o ínicio de um loop que podemos chamar como quisermos, que nesse caso é chamado de loop\_label. A partir de agora quando utilizarmos o comando ITERATE seguido do nome que demos a esse loop, o código irá voltar para esse ponto inicial, criando um loop.

Depois disso vemos um comando IF, que é similar a forma que é utilizado em python. Depois do IF nós escolhemos uma comparação que, se for verdadeira, executará uma parte específica do código. Nesse caso se y1 for maior que max\_num\_1, executaremos o comando LEAVE loop\_label, que encerraria o nosso loop.

Então declaramos uma nova variável, chamada de countage, que também tem o valor igual a um query. Esse query é similar ao visto anteriormente, mas nesse caso, está contando o número de vezes que a user\_id com valor igual a y1 aparece na tabela de empréstimo, ou seja, quantos livros o usuário y1 pegou emprestado.

Agora que temos o número de livros retirados pelo usuário, modificamos a tabela user utilizando UPDATE, de forma que na linha na qual o usuário tem a identidade de número y1, escrevemos o valor que está armazenado na variável countage.

Para finalizar o loop nós adicionamos 1 ao valor de y1 e então usamos ITERATE para reiniciar o loop. Na linha abaixo, encerramos o loop e então encerramos o procedimento.

Então vimos que esse procedimento pega o número total de usuários do sistema, e realiza um número igual de iterações. Em cada iteração o usuário de número y1 tem o seu número de livros contado e armazenado na coluna borrow\_number, e no final y1 incrementado em 1. Isso ocorre repetidamente até que y1 seja maior do que max\_num\_1, e nesse momento o procedimento é encerrado.

Se executarmos esse código, teremos então criado esse procedimente. Para executar o procedimento depois de cria-lo, basta utilizar o comando CALL.

CALL check\_borrow();

Depois de executarmos nosso CALL, podemos selecionar a tabela user, e veremos que o número presente na coluna borrow\_number de cada usuário, agora será consistente com os valores presentes na tabela empréstimo. Então sempre que quisermos garantir que esses números estão corretos é só executar o CALL novamente.

Vale a pena ressaltar que esses procedimentos ficam armazenados no DB, então apesar de termos criado eles no Workbench, se quisermos podemos realizar o CALL no PopSQL e ele funcionará corretamente.

Outro problema do nosso DB é que a coluna late da tabela user, que deveria indicar quando o usuário está com 3 livros emprestados ou está atrasado na devolução de um livro, também não tem um mecanismo que garanta a veracidade dessa informação. Logo, vamos criar um procedimento que irá checar se alguma dessas condições é verdadeira, e colocará o valor True na coluna caso seja verdade.

delimiter $$

CREATE PROCEDURE block\_user()

BEGIN

    SET @max\_num\_2 = (SELECT count(user\_id) FROM user);

    SET @x1 = 1;

    loop\_label: LOOP

        IF @x1 > @max\_num\_2 THEN

            LEAVE loop\_label;

        END IF;

        UPDATE user

        JOIN loan

        ON loan.user = user.user\_id

        SET late = IF( user.borrow\_number > 2 OR loan.due\_date < CURRENT\_DATE, True , False)

        WHERE user\_id = @x1;

UPDATE user

        SET late = IF(borrow\_number = 0, False, late)

        WHERE user\_id = @x1;

        SET @x1 = @x1 + 1;

        ITERATE loop\_label;

    END LOOP loop\_label;

END$$

delimiter ;

Esse procedimento é bem similar ao visto anteriormente. Criamos duas variáveis, x1 e max\_num\_2, que tem os mesmos valores de y1 e max\_num\_1, estabelecemos um loop com o mesmo nome, e estabelecemos a mesma condição para o fim do loop.

Depois disso fazemos o primeiro UPDATE, que precisará incluir um JOIN entre user e loan, para termos acesso a todas as colunas que vamos precisar. Então designamos à coluna late um valor, igual a um query, na linha na qual o usuário tem o número x1.

Esse query usa um tipo de IF diferente dos outros e funciona da seguinte forma:

IF(condição, resultado se for verdadeiro, resultado se for falso)

Ou seja, nesse caso, se o borrow\_number for maior do que 2 OU a data de devolução do livro for menor do que a data atual, late passará a ser igual a True, e caso nenhuma dessas condições seja verdadeira late passará a ser igual a False. Isso corresponde as duas condições que desejávamos, porém ainda precisamos de mais uma parte para garantir que os valores estejam sempre corretos.

O segundo UPDATE é similar ao primeiro, só que ele apenas checa se o borrow\_number é igual a zero, e se for, designa False como valor de late. Se não for igual a zero, o valor de late não é alterado.

Esse comando é importante pois se alguém pegasse um livro, e atrasasse na sua entrega, esse usuário ficaria marcado como late. Depois que ele devolvesse o livro, o seu número seria removido da lista de empréstimos.

O problema disso é que se ele não estiver na lista de empréstimos, o primeiro processo de checagem não conseguirá encontra-lo e, portanto, não poderá ver que deve alterar o valor de late. Já que esse problema só ocorre quando o usuário está sem nenhum livro emprestado, e se ele não está com um livro emprestado não teria razão para o valor late ser True podemos fazer essa segunda checagem que resolve exclusivamente esse erro.

Depois disso simplesmente incrementamos x1 e iteramos o LOOP como fizemos no procedimento check\_borrow().

Agora imagine que um usuário acabou devolver um livro, e que a biblioteca permita que os livros sejam retornados em qualquer filial, sem precisar ser a qual o livro pertencia originalmente. Precisamos de um jeito de remover esse livro da lista de livros emprestados, e precisamos mudar a informação que indica em qual filial esse livro está localizado.

delimiter $$

CREATE PROCEDURE return\_book(book\_num INT, branch\_num INT)

BEGIN

    DELETE FROM loan

    WHERE book = book\_num;

    UPDATE book

    SET branch = branch\_num

    WHERE book\_id = book\_num;

END$$

delimiter ;

E é exatamente isso que esse procedimento faz. Podemos ver que ele tem uma característica diferente dos que vimos anteriormente, já que existe algo escrito entre os parênteses.

Esses termos entre parênteses são variáveis que serão introduzidas ao mesmo tempo em que o CALL do procedimento for executado. As variáveis são separadas por uma vírgula, e devem ter um nome e o seu tipo de valor especificado, que nesse caso são as variáveis, book\_num e branch\_num, ambos números inteiros.

Depois disso iniciamos o procedimento em si, e deletamos da tabela loan a linha na qual o número do livro é igual a variável book\_num que declaramos ao realizarmos o CALL.

Então realizamos um UPDATE na tabela book, onde o valor de branch passa a ser igual a variável branch\_num, nas linhas onde o número do livro é igual a variável book\_num. Podemos agora chamar esse procedimento sempre que quisermos retirar uma linha da tabela loan.

Agora que sabemos como remover um livro da lista de empréstimos, vamos ver como adicionar um livro a essa lista.

delimiter $$

CREATE PROCEDURE add\_loan1(userx INT, bookx INT)

main\_label:BEGIN

     SET @late\_check= (SELECT late FROM user WHERE user\_id = userx) ;

     SET @error1 = " Usuário em situação irregular.";

     IF @late\_check = 1 THEN

        SELECT @error1;

        LEAVE main\_label;

     END IF;

     SET @CD = CURRENT\_DATE;

     SET @DD = date\_add(current\_date, INTERVAL 30 DAY);

     INSERT INTO loan (start\_date, due\_date, book, user)

     VALUES (@CD, @DD, bookx, userx);

END$$

delimiter ;

Esse procedimento começa declarando as variáveis userx e bookx como números inteiros, que devem ser fornecidos ao realizarmos o CALL. Depois disso, colocamos uma label na seção principal desse procedimento que nos permitirá abortar esse procedimento mais para frente.

Criamos a variável late\_check, que irá armazenar o valor da coluna late, na qual o usuário tem número igual a variável userx, e criamos a variável error1, que usaremos apenas para armazenar uma mensagem.

Então executamos um IF que checa se o valor de late é igual a zero, o que ocorrerá apenas quando late for True. Caso isso ocorra, o procedimento será encerrado pelo comando LEAVE main\_label, e a mensagem que armazenamos em error1 será exibida.

Se o valor não for igual a 1, então o usuário ainda tem permissão para retirar livros, e podemos seguir para o próximo passo. Agora declaramos as variáveis CC e DD, que são a data na qual o livro é retirado, e a data na qual o livro deve ser devolvido respectivamente. Como o livro será emprestado nesse momento, a data de retirada será a data atual, que pode ser obtida usando o comando CURRENT\_DATE.

Como determinamos no planejamento do DB, o prazo de devolução é de 30 dias, então usamos o comando date\_add para adicionar 30 dias à data de hoje. Então usamos a função INSERT INTO para inserir uma nova linha na tabela loan, usando como valores as variáveis CC e DD que acabamos de definir, e as variáveis userx e bookx, que indicam o usuário que está retirando o livro e o livro que está sendo emprestado.

Enquanto o nosso DB é relativamente pequeno, já que foi criado para propósitos ilustrativos, um DB pode ter tabelas com centenas ou milhares de linhas, o que torna útil a existência de uma forma de procurar facilmente os valores que quisermos nas tabelas, então vejamos como podemos fazer um procedimento de busca.

delimiter $$

CREATE PROCEDURE search\_book(search\_term VARCHAR(30))

BEGIN

    SET @search\_term\_2 = NULL;

    SET @search\_term\_2 = (SELECT CAST(search\_term AS UNSIGNED));

    SELECT book.book\_id, book.book\_name, book.genre, author.author\_name, branch.branch\_name, branch.address FROM book

    JOIN author ON author.author\_id = book.author

    JOIN publisher ON publisher.publisher\_id = book.publisher

    JOIN branch ON branch.branch\_id = book.branch

    WHERE (book\_name LIKE CONCAT("%",search\_term,"%")

    OR book.genre LIKE CONCAT("%",search\_term,"%")

    OR branch.branch\_name LIKE CONCAT("%",search\_term,"%")

    OR branch.address LIKE CONCAT("%",search\_term,"%")

    OR author.author\_name LIKE CONCAT("%",search\_term,"%")

    OR book\_id = @search\_term\_2);

END$$

delimiter ;

Esse é um procedimento que faz uma busca na tabela book, por qualquer valor que inclua o termo de busca que digitarmos ao executarmos o CALL.

Primeiro declaramos a variável search\_term\_2 com o valor NULL, para garantir que essa variável sempre esteja vazia no começo do procedimento, o que ajuda a evitar erros. Depois declaramos essa variável novamente, mas dessa vez usamos o comando CAST AS UNSIGNED. Nesse caso, esse comando vai pegar a variável search\_term que introduzimos no CALL, e converte-la em um número inteiro se possível, que vai ser importante para realizar parte da busca.

Depois executamos um SELECT onde selecionamos para exibir as colunas book\_name, genre, book\_id, author\_name, branch\_name e branch.address.e fazemos um JOIN entre as tabelas, book, author, branch e publisher.

Até agora tudo é bem simples, mas o truque está na condição WHERE, na qual nós vamos ligar várias condições utilizando o termo OR, que selecionará todas as linhas que obedeçam a qualquer uma das condições que nós estabelecemos para o WHERE.

A primeira condição do WHERE usa vários comandos novos, o LIKE, o CONCAT, e coringas. O comando LIKE procura por caracteres iguais aos do termo que o seguem, então nesse caso ele irá procurar na coluna book\_name por qualquer texto que seja igual ao search\_term. Porém o search\_term não está sozinho, ele está sendo usado na função CONCAT, junto de dois coringas.

A função CONCAT, realiza o equivalente a concatenação de strings, combinando todos os termos dentro dos parênteses em um só. Então obteremos %search\_term%. Entretanto, o símbolo % equivale a um coringa, que indica um número qualquer de caracteres. Isso significa então que essa condição vai procurar por qualquer valor que contenha o search\_term, independente do que está antes ou depois do search\_term.

Seguimos fazendo isso para todas as colunas que tem o tipo de valor VARCHAR, separando cada condição pelo termo OR. Quando chegamos na parte da condição que envolve uma coluna do tipo INT, basta igualarmos à variável search\_term\_2, que é apenas o nosso search\_term convertido em um número inteiro.

Agora podemos utilizar o CALL com algum termo, como por exemplo CALL(“the”), e o procedimento vai procurar todas as colunas especificadas por qualquer aparição desses caracteres juntos.

Para fechar essa seção vamos criar uma função de busca para a maioria de nossas outras tabelas. Todas elas usam a mesma lógica então não explicarei elas, mas se você entendeu a que acabamos de fazer, irá entender as outras também.

delimiter $$

CREATE PROCEDURE search\_user(search\_term VARCHAR(30))

BEGIN

    SET @search\_term\_1 = NULL;

    SET @search\_term\_1 = (SELECT CAST(search\_term AS UNSIGNED));

    SELECT \* FROM user

    WHERE (first\_name LIKE CONCAT("%",search\_term,"%")

    OR last\_name LIKE CONCAT("%",search\_term,"%")

    OR contact LIKE CONCAT("%",search\_term,"%")

    OR address LIKE CONCAT("%",search\_term,"%")

    OR user\_id = @search\_term\_1

    OR borrow\_number = @search\_term\_1);

END$$

delimiter ;

delimiter $$

CREATE PROCEDURE search\_worker(search\_term VARCHAR(30))

BEGIN

    SET @search\_term\_3 = NULL;

    SET @search\_term\_3 = (SELECT CAST(search\_term AS UNSIGNED));

    SELECT worker.worker\_first\_name, worker.worker\_last\_name, branch.branch\_name, worker.work, worker.worker\_contact, worker.worker\_address, worker.worker\_id FROM worker

    JOIN job ON worker.worker\_id = job.worker\_id

    JOIN branch ON job.branch\_id = branch.branch\_id

    WHERE (worker.worker\_first\_name LIKE CONCAT("%",search\_term,"%")

    OR worker.worker\_last\_name LIKE CONCAT("%",search\_term,"%")

    OR worker.worker\_contact LIKE CONCAT("%",search\_term,"%")

    OR worker.worker\_address LIKE CONCAT("%",search\_term,"%")

    OR worker.work LIKE CONCAT("%",search\_term,"%")

    OR branch.branch\_name LIKE CONCAT("%",search\_term,"%")

    OR worker.worker\_id = @search\_term\_3);

END$$

delimiter ;

delimiter $$

CREATE PROCEDURE search\_publisher(search\_term VARCHAR(30))

BEGIN

    SET @search\_term\_4 = NULL;

    SET @search\_term\_4 = (SELECT CAST(search\_term AS UNSIGNED));

    SELECT \* FROM publisher

    WHERE (publisher\_contact LIKE CONCAT("%",search\_term,"%")

    OR publisher\_name LIKE CONCAT("%",search\_term,"%")

    OR publisher\_id = @search\_term\_4);

END$$

delimiter ;

delimiter $$

CREATE PROCEDURE search\_loan(search\_term VARCHAR(30))

BEGIN

    SET @search\_term\_5 = NULL;

    SET @search\_term\_5 = (SELECT CAST(search\_term AS UNSIGNED));

    SELECT loan.loan\_id,loan.start\_date, loan.due\_date, book.book\_name, book.book\_id, user.last\_name, user.user\_id FROM loan

    JOIN book

    ON loan.book = book.book\_id

    JOIN user

    ON loan.user = user.user\_id

    WHERE (loan.start\_date LIKE CONCAT("%",search\_term,"%")

    OR loan.due\_date LIKE CONCAT("%",search\_term,"%")

    OR book.book\_name LIKE CONCAT("%",search\_term,"%")

    OR user.last\_name LIKE CONCAT("%",search\_term,"%")

    OR loan.loan\_id = @search\_term\_5

    OR book.book\_id = @search\_term\_5

    OR user.user\_id = @search\_term\_5);

END$$

delimiter ;

delimiter $$

CREATE PROCEDURE search\_author(search\_term VARCHAR(30))

BEGIN

    SET @search\_term\_6 = NULL;

    SET @search\_term\_6 = (SELECT CAST(search\_term AS UNSIGNED));

    SELECT \* FROM author

    WHERE (author\_name LIKE CONCAT("%",search\_term,"%")

    OR author\_id = @search\_term\_6);

END$$

delimiter ;

**Integração Entre Python e MySQL**

**4.1 – Definição das Funções Principais**

Agora que sabemos tudo que precisamos saber sobre python e MySQL, precisamos só criar algumas funções que usaremos muitas vezes ao longo do nosso programa.Mas antes disso vamos importar os módulos que usaremos ao longo do programa. Para isso, primeiro vamos abrir o PyCharm, e depois clicar na aba file, e depois em settings. Então precisamos selecionar o nosso python interpreter para esse projeto.

Ao clicar nele, veremos uma lista com todos os módulos instalados, e no topo da lista temos um símbolo + que devemos clicar para adicionar mais módulos. Na barra de busca precisamos procurar por 3 módulos, tabulate, pandas e mysql-connector-python. Procure por cada um individualmente e clique em “Install Package”. No caso do mysql-connector-python, antes de instalar, selecione a opção de instalar uma versão específica, e escolha a versão 8.0.30.

Depois disso basta usarmos os comandos import como vimos anteriormente. Importaremos o conector, e também especificamente a função Error do conector, o comando tabulate do módulo tabulate, e pandas abreviando o nome como pd. Além disso, criamos algumas restrições para as tabelas que vamos gerar utilizando pandas.

import mysql.connector  
from mysql.connector import Error  
from tabulate import tabulate  
import pandas as pd  
pd.options.display.max\_columns = None  
pd.options.display.max\_rows = 100  
pd.options.display.width = 500

Agora podemos começar a criar as nossas funções principais. Nosso primeiro objetivo será conectarmos o PyCharm ao nosso servidor MySQL e ao nosso DB, então vamos criar uma função para isso.

def create\_db\_connection(host\_name, user\_name, user\_password, db\_name):  
 connection = None  
 try:  
 connection = mysql.connector.connect(  
 host=host\_name,   
 user=user\_name,   
 passwd=user\_password,  
 database=db\_name   
 )  
 print("MySQL Database connection successful")  
 except Error as err:  
 print(f"Error: '{err}'")

Primeiro declaramos o nome da nossa função e os valores que devem ser fornecidos ao chamar a função. A primeira coisa que devemos fazer é declarar a variável connection com o valor None, pois isso garante que toda vez que essa função for chamada essa variável vai iniciar vazia. Então utilizamos o comando try/except para lidar com erros que possam acontecer durante esse processo.

Dentro da seção Try, redeclaramos connection como sendo igual ao comando mysql.connector.connect, que é um comando próprio do módulo que realiza a conexão ao servidor. Para que funcione corretamente precisamos fornecer os valores de host, user, passwd e database. Dentro da função deixaremos esses valores como variáveis, os valores reais só serão introduzidos quando chamarmos a função.

Caso a conexão funcione, o programa irá exibir uma mensagem de sucesso e estaremos conectados ao servidor. Se ocorrer algum erro do tipo Error, o programa irá exibir a mesma mensagem de erro que apareceria caso um query falhasse no PopSQL. Podemos testar esse comando agora.

connection = create\_db\_connection("localhost", "root", "sua\_senha", "nome\_do\_DB")

Agora que estamos conectados, precisamos poder ler nosso DB e também modificar nosso DB.

def execute\_query(connection, query):  
 cursor = connection.cursor()  
 try:  
 cursor.execute(query)  
 connection.commit()  
 print("Query successful")  
 except Error as err:  
 print(f"Error: '{err}'")

Essa é a função que usaremos para realizar qualquer comando que modifique nosso DB, que chamaremos de execute\_query, e especificamos que devemos fornecer tanto o valor connection quanto o valor query.

Criamos a variável cursor, que igualamos ao comando connection.cursor(), que funcionará como se estivéssemos escrevendo na linha de comando que usamos para criar nosso DB inicialmente. A variável connection indica que esse cursor deve estar acessado a conexão que especificamos anteriormente.

Então usamos a mesma exceção que usamos anteriormente, e especificamos que nosso cursor deve executar o query que inserirmos nessa função, e utilizamos o connection.commit() para especificar que queremos alterar o DB, e exibimos uma mensagem de sucesso.

def read\_query(connection, query):  
 cursor = connection.cursor()  
 result = None  
 try:  
 cursor.execute(query)  
 result = cursor.fetchall()  
 return result  
 except Error as err:  
 print(f"Error: '{err}'")

Essa é a função que usaremos para exibir nossas tabelas, ou ler alguma informação do DB. Começamos com o mesmo processo que fizemos anteriormente, com a exceção da variável result que será iniciada sem nenhum valor, para garantir que a variável sempre esteja limpa quando começarmos a executar essa função. Usamos também a mesma exceção anterior, e usamos o cursor para executar nosso query.

Agora ao invés de usar o commit, estamos utilizando o cursor.fetchall que simplesmente pega toda a informação que está guardada no cursor, e armazenamos essa informação na variável result. Por fim usamos o comando return, que retorna o valor de result para o ponto no programa em que essa função foi chamada.

def read\_query\_index(connection, query,x):  
 cursor = connection.cursor()  
 result = None  
 try:  
 cursor.execute(query)  
 result = cursor.fetchone()[x]  
 return result  
 except Error as err:  
 print(f"Error: '{err}'")

Essa função é igual a que vimos anteriormente, com uma pequena diferença. Utilizamos uma variável x, e trocamos o fetchall de antes por um fetchone() [x]. Isso significa que ao invés de colocarmos todos os valores do cursor na variável, nós selecionamos apenas um valor específico e descartamos o resta. Esse valor é definido pelo x, que indica o índice do valor que será selecionado. Por exemplo se selecionarmos uma linha com 10 valores, e o x = 3, essa função retornará apenas o quarto valor presente nessa linha.

Com isso temos todas as nossas funções principais, e podemos começar agora a realizar nossos queries.

**4.2 – Execução de Queries**

Antes de começarmos garanta que os comandos de import, as definições de funções e o comando que inicia a conexão, estejam sempre no topo do nosso código nessa ordem. Isso irá impedir que as ações sejam executadas fora de ordem e criem algum erro.

Para realizarmos queries no python usaremos a mesma lógica que vimos durante a seção de SQL, porém com um pequeno passo a mais.

q1 = """SELECT \* FROM book;"""  
results = read\_query(connection,q1)  
for result in results:  
 print(result)

Como podemos ver, estamos querendo exibir a tabela book, mas ao invés de digitarmos nossos queries diretamente na função read\_query, nós primeiro armazenamos nossos queries em uma variável, nesse caso q1, e inserimos essa variável na função. O query então será feito e a tabela será armazenada na variável results como uma lista de listas, na qual cada elemento dessa lista é uma linha da nossa coluna.

Se printarmos results diretamente vamos obter, todas as linhas da tabela, porém escritas horizontalmente uma atrás da outra, o que causa uma legibilidade terrível. Para isso utilizamos o loop for, no qual declaramos que para cada elemento único dentro de results, queremos executar o print desse elemento. Na prática isso destrincha a lista, exibindo cada linha da tabela em uma linha diferente, e criando uma visualização muito mais legível. Mas essa formatação ainda não é muito boa, então vamos melhorá-la.

q1 = """SELECT \* FROM book"""  
results = read\_query(connection,q1)  
from\_db = []  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
df = pd.DataFrame(from\_db)  
print(df)

Estamos realizando o mesmo query de antes, mas agora antes do for criamos uma lista vazia chamada de from\_db. Dentro do loop for, ao invés de exibir a lista de linha em linha, nós vamos destrinchar a lista como fizemos anteriormente, mas iremos adicionar essas linhas que obtemos diretamente à nossa lista from\_db.

Depois disso definiremos uma variável df, que será igual a nossa lista from\_db depois de passar pelo comando pd.DataFrame. Esse comando é parte o módulo Pandas, e ele tem a função de formatar a lista que fornecemos para ele em uma tabela, que é chamada de dataframe. Por fim, printamos a variável df, e obtemos uma lista com uma formatação muito mais legível. Porém, podemos melhorar ainda mais esse processo, utilizando o módulo tabulate.

q1 = """SELECT \* FROM book"""  
results = read\_query(connection,q1)  
from\_db = []  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Book ID", "Book Name", "Genre", "Branch ID", "Author", "Publisher ID"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))

Utilizamos o mesmo query novamente, e fazemos os mesmos passos até o fim do loop for. Nesse momento definimos a lista columns, que deve ser composta pelos nomes que queremos dar para as colunas da nossa lista, na ordem na qual essas colunas aparecem na lista. Agora ao definirmos a variável df, devemos definir que as colunas da nossa tabela, devem utilizar os nomes da lista columns, utilizando o comando columns = columns.

Por fim dentro comando print, usamos o comando tabulate no qual especificamos em ordem, o nome do dataframe que usaremos, os nomes das colunas que definimos como ‘keys’, a fonte que será usada na tabela, e se devemos mostrar o índice na tabela, que geralmente não precisamos.

Agora nossas tabelas estão perfeitamente formatadas, e precisaremos escolher alguma dessas opções sempre que quisermos exibir alguma tabela.

Para mostrar como é fácil executar esses queries, vamos pegar um dos queries mais complexos que realizamos anteriormente e ver o resultado.

q1 = """SELECT book\_name, genre, author\_name, branch.branch\_name, branch.address, publisher.publisher\_name FROM book  
 JOIN author  
 ON book.author = author.author\_id  
 JOIN branch  
 ON book.branch = branch.branch\_id  
 JOIN publisher  
 ON book.publisher = publisher.publisher\_id  
 WHERE branch.branch\_name = "Branch A" OR branch.branch\_name = "Branch C"  
 ORDER BY book.genre DESC;"""  
results = read\_query(connection,q1)  
from\_db = []  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Book Name", "Genre", "Author", "Branch", "Address", "Publisher"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))

Como podemos ver podemos mudar o query simplesmente alterando o valor da variável q1, usando exatamente a mesma sintaxe que utilizamos no PopSQL, e para exibir a lista só precisamos alterar nossa lista columns. Todo o resto do código pode ser mantido igual.

Vamos agora alterar alguma informação dentro das nossas tabelas.

q1 = """UPDATE user  
 SET last\_name = "Jackson"  
 WHERE user\_id = 12  
 """  
q2 = """SELECT \* FROM user;"""  
  
execute\_query(connection,q1)  
results = read\_query(connection,q2)  
from\_db = []  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["User ID", "First Name", "Last Name", "Contact","Borrow Number", "Address", "Late"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))

Nesse exemplo, temos um query que altera o sobrenome do usuário 12, para “Jackson”. Note que precisamos usar as aspas para mostrar que esse é um valor VARCHAR. Em seguida fazemos um query para ler a tabela de usuários e ver se a mudança ocorreu corretamente.

Com isso, aprendemos a ler nossas tabelas, extrair informações específicas dela, e aprendemos a fazer modificações no nosso DB.

**4.3 – Código de Exemplo Explicado**

Para demonstrar a combinação do SQL e Python, criarei um código que funcionará como uma forma bem básica de interface visual, que simulará um sistema que poderia ser utilizado pelos funcionários administrativos dessa biblioteca.

Nosso sistema nos deixará exibir qualquer uma de nossas tabelas principais, adicionar e remover linhas à nossa lista de empréstimos, cadastrar novos usuários, remover usuários, modificar informações sobre usuários, adicionar ou remover livros à lista de livros, permitir a realização de buscas em todas as nossas tabelas, além de manter o número de livros de cada usuário e a situação da permissão de cada usuário, constantemente atualizados.

Por fim, todas essas ações devem poder ser executados por uma pessoa leiga, sem nenhum conhecimento sobre python ou SQL. Isso pode parecer difícil, mas graças aos procedimentos e funções que criamos, será muito mais fácil do que parece. O arquivo python que contém o código inteiro pode ser encontrado na seção de Anexos.

Nosso código deve começar da mesma forma que fizemos durante esses últimos queries, com os módulos importados no topo, seguidos das definições de funções, e do estabelecimento da conexão, tudo isso antes de começarmos o resto do programa.

Como esse sistema deve ser usado por pessoas leigas, queremos que ele esteja sempre rodando, e que um pequeno erro não cause o encerramento do nosso programa. Isso significa então que precisamos utilizar o loop while e o try/except.

while True:  
   
 try:  
   
   
 except ValueError:

print("An error has ocurred!")

Esse será o esqueleto do nosso código, com o loop while que se repetirá eternamente, e a maior parte do código será indentado dentro do Try, para protegê-lo de erros. Mas antes de entrar nos detalhes do código vamos estabelecer algumas ações que queremos realizar toda vez que o loop while é reiniciado.

while True:  
 connection.reconnect()  
 branch = 4  
 q\_check\_borrow = """CALL check\_borrow();"""  
 q\_block\_user = """CALL block\_user();"""  
 execute\_query(connection, q\_check\_borrow)  
 execute\_query(connection, q\_block\_user)  
   
 try:  
  
  
 except ValueError:

print("An error has ocurred!")

A primeira ação é um comando reconnect que usamos aqui para reduzir a chance da perda de conexão com o DB. Depois definimos o número da filial hipotética que está usando esse programa, pois criando essa variável podemos atrelar todas as ações do código especificamente a filial em questão, e só precisaríamos modificar esse número para customizar nosso programa para cada uma das outras filiais.

Em seguida armazenamos dois queries em duas variáveis, e depois executamos esses queries. Como podemos ver esses queries são os procedimentos check\_borrow e block\_user que criamos anteriormente na seção de SQL. Então o que esses comandos fazem é que, a cada loop while que ocorre, o programa vai automaticamente atualizar o número de livros com cada usuário, e irá atualizar o valor que indica se o usuário ainda tem permissão para retirar livros.

Agora todo o resto do código estará indentado dentro do Try, que minimizará o efeito de erros.

O nosso sistema funcionará constantemente, mas como queremos que uma pessoa possa interagir com ele, precisamos de alguma forma de pausar o programa para esperar que a interação seja realizada.

option = int(input("""  
SELECT OPTION:  
  
1- Lists  
2- Search  
3- Loan Book  
4- Return Book  
5- Manage User Account  
6- Manage Books  
  
"""))

Esse commando faz exatamente isso. Primeiro nós criamos um input, que irá exibir a mensagem entre aspas, continuamente, até que algum valor seja inserido no sistema. Isso na prática dará ao programa uma funcionalidade similar a de um menu, no qual o usuário poderá ver todas as suas opções enquanto escolhe a opção desejada.

Para selecionar uma opção, basta o usuário digitar o número que está indicado ao lado da opção desejada, que será então armazenado na variável option. Como temos 6 opções nosso programa irá agora se dividir em 7 caminhos, um para cada um dos números que podem ser selecionados, e um para caso o número digitado não esteja entre as opções.

A melhor forma de estruturarmos essa escolha é utilizando um comando IF, da seguinte forma:

if option == 1:  
   
elif option == 2:  
   
elif option == 3:  
   
elif option == 4:  
   
elif option == 5:  
  
elif option == 6:  
   
else:  
 print("Escolha Um Numero Valido")

Cada possível input do usuário selecionará um desses IFs diferentes, e caso o número não esteja disponível, exibiremos o aviso visto em else. Agora vamos ver então o processo que ocorre dentro de cada um desses IFs, e de cada outro IF que possa estar dentro deles.

Isso significa que a estrutura passará a ser muito mais complexa e um pouco difícila de acompanhar, por isso sugiro que realize o download do código inteiro para poder acompanhar mais facilmente. Além disso antes de cada seção irei especificar em que parte do código estamos localizados, usando o seguinte método:

While > try > if option == 1 > if option\_1 == 1

Nesse exemplo, o Código estará localizado dentro do loop while, dentro do try, dentro do if option == 1, e dentro do if option\_list == 1. Isso deve deixar um pouco mais clara a estrutura geral do programa. Por razões de praticidade, irei sempre omitir o while e o try dessa sequência, já que como mencionei anteriormente, todo o resto do código estará dentro deles.

**if option == 1 >**

Se estamos nesse ponto, significa que o usuário quer visualizar as tabelas que compõe o nosso DB, então para isso, devemos exibir um novo menu, com todas as opções de tabela disponíveis.

q1 = """SELECT loan.start\_date, loan.due\_date, loan.book, book.book\_name, loan.user   
 FROM loan   
 JOIN book  
 ON loan.book = book.book\_id"""  
q2 = """SELECT \* FROM book"""  
q3 = """SELECT \* FROM user"""  
q4 = """SELECT \* FROM worker"""  
q5 = """SELECT \* FROM branch"""  
q6 = """SELECT \* FROM author """  
q7 = """SELECT \* FROM publisher"""  
q8 = """SELECT \* FROM job"""  
  
option\_list = int(input("""  
SELECT OPTION:   
  
1- Loan List  
2- Book List  
3- User List  
4- Worker List  
5- Branch List  
6- Author List  
7- Publisher List  
8- Job List  
  
"""))

Essa configuração, será igual àquela que vimos anteriormente na qual exibimos as opções para que o usuário escolha qual delas ele quer visualizar, mas nesse caso antes de exibirmos as opções nós declaramos 8 variáveis, uma para cada opção.

Cada uma dessas variáveis é um query, e são todas comandos SELECT simples, com a exceção de q1 que inclui o uso de um comando JOIN. Como queremos apenas visualizar as tabelas, esses queries fazem tudo que precisamos, e podemos então abrir mais 8 IFs, um para cada opção, e mais um else, para caso o número digitado pelo usuário não esteja disponível. Vamos ver agora o que está dentro do primeiro desses 8 novos IFs.

**if option == 1 > if option\_list == 1 >**

Se estamos nesse ponto do programa, o usuário pediu para exibirmos a tabela loan da forma especificada no query da variável q1, então só precisamos efetuar o método para leitura de query que vimos anteriormente.

from\_db = []  
results = read\_query(connection, q1)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Start Date", "Due Date", "Book ID", "Book Name", "User ID"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

Todo esse código já foi explicado anteriormente, é o processo padrão para quando queremos exibir tabelas com boa legibilidade, com a exceção de um pequeno elemento novo no final.

Esse input, com a explicação “Press Any Key” é utilizado para realizar uma pausa no programa, pois precisamos que o programa esteja parado para que o usuário possa ler a lista calmamente. Isso nos permite fazer essa pausa, e quando o usuário apertar qualquer botão, o programa então terá permissão para continuar.

Como atendemos ao desejo do usuário esse é o fim desse caminho específico dentro do código, o que significa que o usuário será enviado de volta para o início do loop While, onde ele poderá então selecionar outras opções.

Todos os outros IFs dentro de option == 1 serão idênticos, com a excessão da lista columns, que será única para cada um, logo apenas mostrarei o código deles aqui sem explicação.

**if option == 1 > if option\_list == 2 >**

from\_db = []  
results = read\_query(connection, q2)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Book ID", "Book Name", "Genre", "Branch ID", "Author", "Publisher ID"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

**if option == 1 > if option\_list == 3 >**

from\_db = []  
results = read\_query(connection, q3)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["User ID", "First Name", "Last Name", "Contact", "Books Borrowed", "Address", "Late"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

**if option == 1 > if option\_list == 4 >**

from\_db = []  
results = read\_query(connection, q4)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Worker ID", "First Name", "Last Name", "Contact", "job", "Weekly Hours", "Address"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

**if option == 1 > if option\_list == 5 >**

from\_db = []  
results = read\_query(connection, q5)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Branch ID", "Address", "Name"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

**if option == 1 > if option\_list == 6 >**

from\_db = []  
results = read\_query(connection, q6)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Author ID", "Name"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

**if option == 1 > if option\_list == 7 >**

from\_db = []  
results = read\_query(connection, q7)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Publisher ID", "Name", "Contact"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

**if option == 1 > if option\_list == 8 >**

from\_db = []  
results = read\_query(connection, q8)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Worker ID", "Branch ID"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
Press Any Key  
""")

**if option == 1 > else >**

print("Escolha Um Numero Valido")

**elif option == 2 >**

Se estamos nesse ponto do programa, o usuário quer realizar uma busca em alguma das nossas tabelas, então precisamos exibir mais um menu para que o usuário selecione em qual lista a busca irá ocorrer.

option\_2 = int(input("""   
SELECT OPTION:   
  
1- Search Book  
2- Search User  
3- Search Worker  
4- Search Publisher  
5- Search Loans  
6- Search Author  
  
"""))

Como temos 6 opções, precisamos criar mais 6 IFs, e um else, nos quais iremos realizar o método de busca dentro da tabela especificada.

**elif option == 2 > option\_2 == 1 >**

search = input("""  
Search: """)  
q1 = f"""CALL search\_book("{search}");"""  
from\_db = []  
results = read\_query(connection, q1)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Book Id", "Name", "Genre", "Author", "Branch", "Branch Address"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
  
Press any key  
  
""")

A primeira coisa que fazemos é pedir que o usuário escreva algum termo para utilizarmos em nossa busca, que é então armazenado na variável search, do tipo string. Em seguida nós criamos uma nova variável, que contém um query.

Como podemos ver o query em questão está utilizando o procedimento que criamos anteriormente, o search\_book, que faz uma busca por um termo na tabela book. Repare também que na parte do procedimento em que devemos inserir o termo de busca, estamos realizando uma concatenação de strings, usando a f-string, que está inserindo o valor da variável search no lugar do termo de busca do nosso procedimento.

Na prática isso é como se digitássemos o nosso procedimento “CALL search\_book(x);” dentro do SQL, onde o x é igual ao que foi digitado pelo usuário inicialmente. O resto dessa seção, é apenas o método de exibir as tabelas que vimos muitas vezes anteriormente.

Mais uma vez, todos os IFs serão idênticos, com a exceção da variável columns, e nesse caso, do query q1, que em cada caso irá chamar um dos procedimentos de busca que criamos na seção de SQL.

**elif option == 2 > option\_2 == 2 >**

search = input("""  
Search: """)  
q1 = f"""CALL search\_user("{search}");"""  
from\_db = []  
results = read\_query(connection, q1)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["User ID", "First Name", "Last Name", "Contact", "Books Borrowed", "Address", "Late"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
  
Press any key  
  
""")

**elif option == 2 > option\_2 == 3 >**

search = input("""  
Search: """)  
q1 = f"""CALL search\_worker("{search}");"""  
from\_db = []  
results = read\_query(connection, q1)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["First Name", "Last Name", "Workplace", "Job", "Contact", "Address", "ID"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
  
Press any key  
  
""")

**elif option == 2 > option\_2 == 4 >**

search = input("""  
Search: """)  
q1 = f"""CALL search\_publisher("{search}");"""  
from\_db = []  
results = read\_query(connection, q1)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Publisher ID", "Name", "Contact"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
  
Press any key  
  
""")

**elif option == 2 > option\_2 == 5 >**

search = input("""  
Search: """)  
q1 = f"""CALL search\_loan("{search}");"""  
from\_db = []  
results = read\_query(connection, q1)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Loan ID", "Start Date", "Due Date", "Book Name", "Book ID", "Last Name", "User ID"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
  
Press any key  
  
""")

**elif option == 2 > option\_2 == 6 >**

search = input("""  
Search: """)  
q1 = f"""CALL search\_author("{search}");"""  
from\_db = []  
results = read\_query(connection, q1)  
for result in results:  
 result = list(result)  
 from\_db.append(result)  
columns = ["Author ID", "Name"]  
df = pd.DataFrame(from\_db, columns=columns)  
print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
input("""  
  
Press any key

""")

**elif option == 2 > option\_2 == else >**

print("Escolha Um Numero Valido")

**elif option == 3 >**

Se estamos nessa parte do programa então queremos realizar o empréstimo de um livro, e registrar esse empréstimo na nossa tabela loan.

book\_id = int(input("""Book ID: """))  
user\_id = int(input("""User ID: """))  
q01 = f"""SELECT COUNT(book\_id) from book WHERE book\_id = {book\_id};"""  
q02 = f"""SELECT late FROM user WHERE user\_id = {user\_id};"""  
q03 = f"""SELECT COUNT(user\_id) from user WHERE user\_id = {user\_id};"""  
q1 = f"""CALL add\_loan1({user\_id},{book\_id});"""  
q2 = """SELECT book.book\_name, user, start\_date, due\_date FROM loan  
 JOIN book  
 ON loan.book = book.book\_id;"""  
  
results2 = read\_query\_index(connection, q01, 0)  
book\_check = int(results2)  
results3 = read\_query\_index(connection, q03, 0)  
user\_check = int(results3)  
if user\_check == 0:  
 print("No User With This ID")  
 input("""Press Any Key""")  
  
elif book\_check == 0:  
 print("No Book With This ID")  
 input("""Press Any Key""")

A primeira coisa que fazemos é perguntar qual livro está sendo emprestado, e qual usuário está retirando o livro. Depois preparamos 5 variáveis, cada uma delas contendo um query, que veremos de um em um.

O q01 conta quantas vezes o número fornecido pelo usuário está presente na lista book, o que na prática retorna 1 caso o livro exista na lista, ou 0 se o livro não existir,

O q02 verifica o valor da coluna late do usuário que está tentando retirar o livro, na prática servindo como verificação da permissão do usuário.

O q03 conta quantas vezes o número do usuário está presente na lista user, que na prática retorna 1 caso o usuário exista na lista, ou 0 se o usuário não existir.

O q1 chama o procedimento “loan\_book1” que criamos anteriormente, e fornece a ele os números do livro e do usuário.

E por fim o q2 exibe a lista loan, para fazermos a verificação de que o empréstimo foi adicionado com sucesso.

Então executamos os queries, q01 e q03, utilizando o “read\_query\_index” com o índice 0, já que só queremos o valor que está no primeiro índice das tabelas que foram obtidas em q01 e q03. Esses valores então colocados nas variáveis book\_check e user\_check.

Esse IF checa se existe um usuário com esse número, e sem não existir, exibe uma mensagem de erro indicando isso. O ELIF faz a mesma coisa, mas com o número do livro. Se nenhuma dessas condições forem verdade, então o usuário e o livro existem, e podemos ir para o processo de adicionar a linha à tabela loan.

**elif option == 3 > else >**

results = read\_query\_index(connection, q02, 0)  
late\_check = int(results)  
if late\_check != 1:  
 execute\_query(connection, q1)  
 from\_db = []  
 results = read\_query(connection, q2)  
 for result in results:  
 result = list(result)  
 from\_db.append(result)  
 columns = ["Title", "User ID", "Borrow Date", "Due Date"]  
 df = pd.DataFrame(from\_db, columns=columns)  
 print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
 input(" Press Any Key ")  
else:  
 print("""Usuario em Situacao Irregular""")  
 input("""Press Any Key""")

Essa parte começa executando o query q02, para verificar a permissão do usuário. Se late estiver com o valor 0, late\_check também terá valor 0, e exibiremos a mensagem “Usuário em Situação Irregular”. Se late tiver valor 1, late\_check terá valor 1, e, portanto, o usuário terá permissão para retirar um livro.

Confirmando a permissão, basta executarmos o query q1, que adicionará a nova linha à tabela loan, e depois realizamos o procedimento padrão para exibir a tabela.

**elif option == 4 >**

Se estamos nessa parte do código então estamos realizando o processo de devolver um livro.

book\_id = int(input("""Book ID: """))  
q0 = f"""SELECT COUNT(book) from loan WHERE book = {book\_id}"""  
q1 = f"""CALL return\_book({book\_id},{branch});"""  
q2 = """SELECT book.book\_name, user, start\_date, due\_date FROM loan  
 JOIN book  
 ON loan.book = book.book\_id;"""  
results = read\_query\_index(connection, q0, 0)  
book\_check = int(results)  
if book\_check == 1:  
 execute\_query(connection, q1)  
 from\_db = []  
 results = read\_query(connection, q2)  
 for result in results:  
 result = list(result)  
 from\_db.append(result)  
 columns = ["Title", "User Name", "Borrow Date", "Due Date"]  
 df = pd.DataFrame(from\_db, columns=columns)  
 print(tabulate(df, headers='keys', tablefmt='psql', showindex=False))  
 input(" Press Any Key ")  
else:  
 print("Book Is Not Loaned")  
 input(" Press Any Key ")

Na primeira linha, pedimos o número do livro que está sendo devolvido. Depois declaramos 3 variáveis que contém um query cada.

O q0 checa se o livro que está sendo devolvido estava presente na lista de empréstimos.

O q1 chama o procedimento return\_book que criamos anteriormente, enquanto fornece o número do livro inserido anteriormente, e o número da filial que pré-definimos no início do programa.

O q2 exibe a tabela loan, para verificarmos que a linha foi removida corretamente.

Depois disso, executamos o query q0, obtendo o valor da resposta ao query, e depositamos esse valor na variável book\_check. Se book\_check for igual a 0, o livro não está na tabela loan, e iremos exibir uma mensagem de erro “Book Is Not Loaned”.

Se book\_check for igual a 1, executamos o query 1, que irá remover o livro da lista de empréstimos, e realizamos o procedimento padrão de exibição de tabela, para verificar se o procedimento foi executado corretamente.

**elif option == 5 >**

Se estamos nesse ponto, significa que queremos fazer alguma modificação à nossa tabela de usuários, então para isso, devemos exibir um novo menu, com todas as opções disponíveis.

option\_5 = int(input("""  
SELECT OPTION:  
  
1 - Add User  
2 - Remove User  
3 - Update User  
  
"""))

Como temos 3 opções, devemos criar 3 Ifs, um para cada jeito diferente de modificar a tabela user.

**elif option == 5 > if option\_5 == 1 >**

user\_borrow\_number = 0  
user\_late = 0  
user\_user\_id = 0  
user\_firstname = input("First Name: ")  
user\_lastname = input("Last Name: ")  
user\_contact = input("Contact Information: ")  
user\_address = input("Address: ")  
q1 = f"""INSERT INTO user VALUES   
 ({user\_user\_id}, "{user\_firstname}", "{user\_lastname}", "{user\_contact}", {user\_borrow\_number},  
 "{user\_address}", {user\_late});  
 """  
execute\_query(connection, q1)  
input("Press Any Key")

O objetivo dessa opção é adicionar um novo usuário ao nosso sistema, o que podemos fazer facilmente, usando um query do tipo INSERT INTO e digitando as informações de uma em uma.

As duas primeiras variáveis, correspondem aos valores de borrow\_number e late, que devem ser zero, pois o usuário está sendo adicionado agora e, portanto, não têm como ter mais de 3 livros retirados ou estar atrasado na devolução de algum livro. O número de usuário será zero também, já que nosso DB usa AUTO\_INCREMENT para a PK da tabela user, ou seja, ao adicionarmos o valor zero, o DB vai automaticamente transformá-lo no próximo número válido para essa coluna.

Depois disso usamos o comando input para obter o nome, sobrenome, telefone de contato, e endereço do usuário, e criamos um query do tipo INSERT INTO dentro da variável q1, utilizando a f-string para inserir os valores que acabamos de obter nos espaços apropriados do comando INSERT.

Por fim só precisamos executar o query q1, utilizando execute\_query.

**elif option == 5 > if option\_5 == 2 >**

user\_user\_id = input("User ID to be Deleted:")  
q1 = f"""DELETE FROM user WHERE user\_id = {user\_user\_id};"""  
execute\_query(connection, q1)  
input("Press Any Key")

O objetivo dessa opção é deletar um usuário da nossa tabela user. Para isso, nós simplesmente usamos a função input para obter o número do usuário que queremos deletar da tabela, e utilizamos um query básico do tipo DELETE FROM usando o número de usuário que obtemos.

Armazenamos esse query na variável q1, e então utilizamos execute\_query com o valor de q1.

**elif option == 5 > if option\_5 == 3 >**

user\_user\_id = int(input("User ID: "))  
option\_5\_1 = input("""  
What Column Should Change?  
\* first\_name  
\* last\_name  
\* address  
\* contact  
""")  
new\_value = input("New Value: ")  
q1 = f"""UPDATE user SET {option\_5\_1} = "{new\_value}" WHERE user\_id = {user\_user\_id}"""  
execute\_query(connection, q1)  
input("Press Any Key")

O objetivo dessa opção é alterar alguma informação sobre um usuário específico, e para isso começamos obtendo o número do usuário desejado usando o comando input. Depois disso, exibimos uma nova lista, que oferece opções de dados que podemos alterar.

Nessa lista fizemos algo um pouco diferente, ao invés de usarmos um número para escolher a opção, podemos digitar a opção que queremos escolher. Depois de escolher a opção, devemos então digitar o novo valor para essa coluna.

Então armazenamos um query na variável q1. Esse query usa uma f-string para inserir as variáveis que acabamos de definir em um comando UPDATE básico, no qual usamos SET para transformar o valor antigo no valor novo, na linha com o número de usuário fornecido anteriormente. Para terminar basta executar esse query com o execute\_query.

**elif option == 5 > if option\_5 == 3 >**

print("Escolha Um Numero Valido")  
input("Press Any Key")

**elif option == 6 >**

Se estamos nesse ponto, significa que queremos fazer alguma modificação à nossa tabela de livros, então para isso, devemos exibir um novo menu, com todas as opções disponíveis.

option\_6\_1 = int(input("""  
 SELECT OPTION:  
  
 1 - Add Book  
 2 - Remove Book  
  
 """))

Como temos 2 opções, devemos criar 2 Ifs, um para cada jeito diferente de modificar a tabela user.

**elif option == 6 > if option\_6\_1 == 1**

book\_book\_id = 0  
book\_name = input("Book Name: ")  
genre = input("Genre: ")  
author\_id = int(input("Author ID: "))  
publisher\_id = int(input("Publisher ID:"))  
q1 = f"""INSERT INTO book VALUES   
 ({book\_book\_id}, "{book\_name}", "{genre}", {branch}, {author\_id},  
 {publisher\_id});  
 """  
execute\_query(connection, q1)  
input("Press Any Key")

Essa opção tem o objetivo de adicionar um livro à nossa tabela book. Começamos declarando a variável book\_id como 0, devido ao AUTO\_INCREMENT, como vimos quando adicionamos usuários à tabela user.

Depois disso, usamos inputs obter os outros valores que compõe uma linha da tabela de livros. Criamos um query do tipo INSERT INTO, usando uma f-string para inserir os valores obtidos nos espaços corretos, e armazenamos esse query em q1. Por fim, basta executar o query com o execute\_query.

**elif option == 6 > if option\_6\_1 == 1**

book\_book\_id = input("Book ID to be Deleted:")  
q1 = f"""DELETE FROM book WHERE book\_id = {book\_book\_id};"""  
execute\_query(connection, q1)  
input("Press Any Key")

Nessa opção, queremos deletar um livro da tabela book. Para isso, simplesmente usamos um input para obter o número do livro que devemos deletar da tabela, e inserimos esse valor em um query do tipo DELETE FROM usando f-strings.

Esse query é armazenado na variável q1, e então executamos o query utilizando a função execute\_query junto de nossa variável q1.

Com isso, o nosso programa está completo, atendendo a todas as necessidades que destacamos no início dessa seção, e perfeitamente utilizável mesmo por uma pessoa que não tenha conhecimento nenhum sobre python e SQL.

Agora execute o código e faça seus próprios testes, pois o melhor jeito de entender o código é mexendo nele, e ao utilizá-lo, podemos ter ideias sobre outros usos ou opções que ele poderia realizar no futuro.

**Anexos**

[Download do Python](https://www.python.org/downloads/) – Baixar a versão que está no topo do site

[Download do PyCharm](https://www.jetbrains.com/pycharm/download/#section=windows) – Baixar a versão mais recente do PyCharm Community

[Download do MySQL](https://dev.mysql.com/downloads/mysql/) – Baixar a versão para o seu sistema operacional