SQL A1 Introdução ao SQL - Criando Bancos de Dados, Tabelas e Inserindo Dados

#SQL #Banco-de-Dados #PostgreeSQL

índice

Introdução ao PostgreSQL

Lançado inicialmente em 1996, o PostgreSQL é um sistema gerenciador de banco de dados relacional (SGBD) de código aberto. Ele utiliza a linguagem SQL (Structured Query Language) para realizar consultas, gerenciar transações e manipular dados, suportando operações CRUD (`CREATE` , `SELECT` , `UPDATE` e `DELETE`).

Conceitos Fundamentais

- **Consulta (Query)**: é uma operação única que solicita informações específicas de um banco de dados, geralmente utilizando o comando SELECT.

```sql
SELECT * FROM clientes WHERE idade > 30;
```

- **Transação (Transaction)**: é uma sequência de operações que são executadas como uma única unidade de trabalho. Ela pode incluir várias consultas e operações de manipulação de dados (como inserções, atualizações ou exclusões). As transações garantem que todas as operações sejam concluídas com sucesso ou, em caso de falha, nenhuma alteração seja aplicada ao banco de dados.
  - A transação pode ser revertida com `ROLLBACK` , garantindo que o banco de dados permaneça consistente (isso é conhecido como **atomicidade**, um dos princípios ACID).

```sql
BEGIN; -- Inicia a transação
UPDATE contas SET saldo = saldo - 100 WHERE id = 1; -- Operação 1
UPDATE contas SET saldo = saldo + 100 WHERE id = 2; -- Operação 2
COMMIT; -- Confirma a transação
```

> **Atenção**
>
> Ao final de cada comando, é necessário adicionar um **ponto e vírgula (;)** para que o SGBD (Sistema de Gerenciamento de Banco de Dados) identifique o término da instrução.

Acessando o banco pela linha de comando

O curso foi projetado para que os comandos sejam executados diretamente no terminal, então primeiro acessamos o banco de dados PostgreSQL a partir do comando: `psql --username=freecodecamp --dbname=postgres`

- `psql` : utilizando para iniciar o terminal interativo do PostgreSQL, uma ferramenta de linha de comando que permite conectar-se a um banco de dados PostgreSQL e executar comandos SQL pelo terminal.
- `--username=freecodecamp` : especifica o nome do usuário que será usado para se conectar ao banco de dados.
- `--dbname=postgres` : define o nome do banco de dados ao qual você deseja se conectar. Aqui, o banco de dados é `postgres` , que é o banco de dados padrão criado automaticamente durante a instalação do PostgreSQL.

Comandos úteis no `psql`

- `\l` : Lista todos os bancos de dados disponíveis.
- `\c nome_do_banco` : Conecta-se a outro banco de dados.
- `\d` : Lista **todas as relações** (tabelas, views, índices, sequências e outros objetos) do banco de dados.
- `\dt` : Lista **apenas todas as tabelas** do banco de dados.
- `\d nome da tabela` : exibe detalhes sobre a estrutura da tabela especificada, incluindo colunas, tipos de dados, restrições e chaves.
- `\q` : Sai do terminal `psql` .

Comandos para Gerenciamento de Banco de Dados

- `CONNECT database_name`: Esse comando faz a mesma função que o `\c` no PostgreSQL ou o `USE database_name` nos SGBDs MySQL e MariaDB. Ele permite conectar-se a um banco de dados específico. É o primeiro comando que precisamos executar para conseguir criar tabelas ou fazer consultas nos dados que estão armazenados nele.
- `CREATE DATABASE database_name`: caso o banco ainda não exista, utilizamos esse comando para cria-lo.
- `ALTER DATABASE database_name ação`: é usado para modificar as propriedades de um banco de dados existente, sendo útil quando é necessário (alguns exemplos):
  - `RENAME TO new_database_name` (renomear o banco de dados).

```
ALTER DATABASE first_database RENAME TO mario_database;
```

  - `OWNER TO new_database_owner` (alterar o proprietário).

```
ALTER DATABASE mario_database OWNER TO luigi;
```

  - `CONNECTION LIMIT limite` (definir ou alterar configurações do banco, como definir um limite de conexões simultâneas).

```
ALTER DATABASE minha_base CONNECTION LIMIT 50;
```

- `DROP DATABASE database_name`: deleta um banco de dados e todas as tabelas e dados dentro dela.

Comandos para gerenciamento de tabelas

Criando e modificando tabelas

- `CREATE TABLE table_name();`: é utilizado para criar uma nova tabela, porém utilizando ele dessa forma criamos a tabela mas não definimos nenhuma coluna ou estrutura, não sendo possível ainda adicionar dados a ela. Para criar uma tabela já definindo as colunas que ela irá ter inicialmente, podemos fazer da seguinte forma:

```
CREATE TABLE clientes (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    data_cadastro DATE DEFAULT CURRENT_DATE
);
```

Caso a tabela seja criada sem colunas, podemos utilizar o `ALTER TABLE` para adiciona-las posteriormente.

- `ALTER TABLE table_name ADD COLUMN column_name DATATYPE CONSTRAINT`: essa é a sintaxe utilizada para alterar uma tabela adicionando uma nova coluna a ela.

```
ALTER TABLE characters ADD COLUMN name VARCHAR(30) NOT NULL;
ALTER TABLE more_info ADD COLUMN birthday DATE;
```

- `ALTER TABLE table_name DROP COLUMN column_name;`: utilizado para remover uma coluna de uma tabela, fazendo com que todos os dados armazenados sejam excluídos também.
  - Caso a coluna tenha dependências ou esteja sendo referenciada por outros objetos no banco de dados, como ser chave estrangeira de alguma outra tabela, ao tentar executar o comando o PostgreSQL retornará um erro. Essas dependências precisarão ser removidas ou alteradas para conseguir excluir a coluna.

```
ALTER TABLE planet_mission DROP COLUMN planet_mission_name;
```

- `ALTER TABLE table_name RENAME COLUMN column_name TO new_name;`: utilizado para alterar o nome de uma tabela

```
ALTER TABLE more_info RENAME COLUMN height TO height_in_cm;
```

- `ALTER TABLE table_name DROP CONSTRAINT constraint_name;`: é usado para remover uma **restrição (constraint)** de uma tabela existente.

- As **restrições** são regras aplicadas às colunas ou à tabela para garantir a integridade dos dados, como chaves primárias, chaves estrangeiras, valores únicos (`UNIQUE`), verificações (`CHECK`), etc.
- Caso a chave primária tenha uma referência em outra tabela, será necessário remover essa relação primeiro antes de executar a remoção.

```
ALTER TABLE planet DROP CONSTRAINT fk_star_id;
```

- `ALTER TABLE table_name ALTER COLUMN column_name TYPE new_data_type;` : utilizada para modificar o tipo de dado que é armazenado em uma coluna.
  - Se os dados existentes na coluna não forem compatíveis com o novo tipo de dado, o PostgreSQL retornará um erro ao tentar executar o comando.
  - Ao alterar para um tipo de dado com menor capacidade (por exemplo, de `VARCHAR(100)` para `VARCHAR(50)`), os dados que excederem o novo tamanho serão truncados ou causarão erros.
  - Se a coluna tiver restrições (como `NOT NULL` , `DEFAULT` , etc.), elas serão mantidas após a alteração do tipo de dado. No entanto, se a restrição não for compatível com o novo tipo de dado, você precisará removê-la antes e recriá-la depois.
- `DROP TABLE table_name` : exclui a tabela e todos os dados contidos nela.

Tipos de Dados (`DATATYPE`) e Restrições (`CONSTRAINT`)

Ao projetar um banco de dados, a seleção dos **tipos de dados** (*datatypes*) e a implementação adequada de **restrições** (*constraints*) são fundamentais para garantir a **integridade**, **eficiência** e **consistência** das informações armazenadas.

Os **tipos de dados** definem o formato dos valores que podem ser armazenados em uma coluna. Por exemplo, é possível definir se uma coluna aceitará números inteiros, textos, datas, valores booleanos ou valores decimais. A escolha correta do tipo de dados não apenas assegura que os valores armazenados sejam apropriados para o contexto, mas também influencia diretamente o desempenho do banco de dados, otimizando o uso de espaço em disco e a velocidade das operações.

Neste módulo, vimos alguns dos tipos de dados mais utilizados em bancos de dados:

Tipo de Dado	Descrição	Exemplo de Uso
INT	Usado para armazenar valores inteiros, tanto positivos quanto negativos. Geralmente utilizado em campos que representam quantidades (como a idade de uma pessoa) ou identificadores (como chaves primárias).	`ALTER TABLE table_name ADD COLUMN idade INT`
VARCHAR(n)	Armazena strings de caracteres com comprimento variável, até um limite definido. Ideal para campos de texto com um tamanho previsível, como nomes, endereços ou descrições.	`ALTER TABLE table_name ADD COLUMN nome VARCHAR(50)`
TEXT	Semelhante ao VARCHAR, mas sem limite máximo de comprimento. Utilizado para armazenar grandes blocos de texto, como descrições longas ou comentários. A principal diferença entre TEXT e VARCHAR é que o TEXT não requer a definição de um limite de caracteres, enquanto o VARCHAR exige especificar esse limite.	`ALTER TABLE table_name ADD COLUMN descricao TEXT`
BOOLEAN	Armazena valores lógicos, representando verdadeiro (`TRUE` = 1) ou falso (`FALSE` = 0).	`ALTER TABLE table_name ADD COLUMN ativo BOOLEAN`
SERIAL	Tipo de dado inteiro que é automaticamente incrementado a cada nova inserção. Frequentemente usado em colunas de chaves primárias.	`ALTER TABLE table_name ADD COLUMN id SERIAL PRIMARY KEY`
DATE	Destinado ao armazenamento de datas no formato `YYYY-MM-DD` .	`ALTER TABLE table_name ADD COLUMN data_nascimento DATE`
NUMERIC(p,e)	Permite armazenar números com precisão decimal, onde `p` define o número total de dígitos e `e` define a quantidade de dígitos à direita da vírgula decimal. Exemplo: `preco NUMERIC(10, 2)` - o número 10 indica o total de dígitos que podem ser armazenados, enquanto o 2 representa o número de dígitos após a vírgula. Isso permite armazenar valores como `12345678.99` , onde até 8 dígitos podem estar antes da vírgula e 2 dígitos após.	`ALTER TABLE table_name ADD COLUMN preco NUMERIC(10, 2)`

Já as restrições são regras aplicadas às colunas ou tabelas para validar os dados inseridos, evitando entradas incorretas ou inválidas. Por exemplo:

2. `NOT NULL` : Garante que uma coluna não aceite valores nulos, ou seja, cada novo registro inserido na tabela deverá obrigatoriamente conter um valor nessa coluna.
3. `UNIQUE` : Garante que todos os valores em uma coluna sejam únicos, ou seja, não haja duplicatas.
4. `PRIMARY KEY` : Definir uma coluna como chave primária garante que cada valor seja único, dessa forma ele torna possível identificar unicamente cada registro em uma tabela utilizando essa coluna.
5. `FOREIGN KEY` : Estabelece uma relação entre duas tabelas, garantindo que o valor em uma coluna corresponda a um valor existente na chave primária de outra tabela.

Inserção e Manipulação de Dados

- `INSERT INTO table_name(column_1, column_2) VALUES(value_column_1, value_column_2);` : utilizado para isenrir novos registros/dados/linha (rows) numa tabela.
  - **Se a tabela tiver outras colunas que não foram mencionadas no comando, elas serão preenchidas com:**
    - **Valores padrão** (se um valor padrão foi definido para a coluna).
    - `NULL` (se a coluna permitir valores nulos).
    - Um **erro** será gerado se a coluna não permitir valores nulos e nenhum valor padrão for definido
  - Se a tabela tiver uma coluna de chave primária com auto-incremento, você não precisa incluí-la no comando INSERT INTO.

```
INSERT INTO characters(name, homeland, favorite_color) VALUES
('Mario', 'Mushroom Kingdom', 'Red'),
('Luigi', 'Mushroom Kingdom', 'Green');
```

- `UPDATE table_name SET column_name = new_value WHERE condition;` : é usado para **modificar dados existentes** em uma tabela. Ele permite alterar os valores de uma ou mais colunas em registros que atendem a uma condição específica.
  - Apenas os registros que atendem a essa condição serão modificados. Se a cláusula `WHERE` não for incluida, **todos os registros da tabela serão atualizados.**

```
UPDATE characters SET favorite_color = 'Orange' WHERE name = 'Daisy';
UPDATE characters SET name = 'Toad' WHERE favorite_color = 'Red';
```

- `DELETE FROM table_name WHERE condition;` : é usado para remover registros (linhas) de uma tabela. Apenas os registros que atendem a essa condição serão removidos, se a cláusula `WHERE` não for incluída, **todos os registros da tabela serão excluídos.**

```
DELETE FROM second_table WHERE username = 'Luigi';
```

Queries de Consultas (`SELECT`)

> `Query` é o nome dado aos comandos de consulta utilizados para interagir com os dados armazenados em tabelas de um banco de dados, p

O `select` é utilizado para consultar os dados dentro de uma ou mais tabelas, permitindo buscar registros específicos ou todos que estão armazenados, de acordo com as condições definidas na query.

- Selecionando todas as colunas e dados da tabela: `SELECT * FROM table_name;`
- Selecionando colunas especificas: `SELECT column_name1, column_name2 FROM table_name;`

Para filtrar os dados ou definir condições utilizamos a cláusula `WHERE`, definindo quais registros devem ser incluídos no resultado.

- `SELECT * FROM table_name WHERE condição;`

Por exemplo, temos a tabela ``**pokemons** abaixo:

Número	Nome	Tipo	Evolução	HP
001	Bulbasaur	Grama/Veneno	Ivysaur	45
004	Charmander	Fogo	Charmeleon	39
007	Squirtle	Água	Wartortle	44
025	Pikachu	Elétrico	Raichu	35
052	Meowth	Normal	Persian	40
129	Magikarp	Água	Gyarados	20
147	Dratini	Dragão	Dragonair	41
151	Mew	Psíquico	-	100
158	Totodile	Água	Croconaw	50

Quando utilizamos o comando `SELECT * FROM pokemons;`, o SGBD retornará todos os dados e colunas da tabela `pokemons`, como mostrado na tabela acima. No entanto, se desejamos que o resultado inclua apenas as colunas `Nome` e `Tipo`, precisamos especificar essas colunas no comando, da seguinte forma: `SELECT Nome, Tipo FROM pokemons;`

Nome	Tipo
Bulbasaur	Grama/Veneno

Nome	Tipo
Charmander	Fogo
Squirtle	Água
Pikachu	Elétrico
Meowth	Normal
Magikarp	Água
Dratini	Dragão
Mew	Psíquico
Totodile	Água

Se quisermos filtrar ainda mais os dados para responder perguntas como **"Quais Pokémons possuem HP maior que 40?"** ou **"Quais Pokémons são do tipo Água?"**, utilizamos a cláusula `WHERE` .

```
SELECT Nome, HP FROM pokemons WHERE HP > 40;
```

Nome	HP
Bulbasaur	45
Squirtle	44
Dratini	41
Mew	100
Totodile	50

```
SELECT Nome, Tipo FROM pokemons WHERE Tipo = 'Água';
```

Nome	Tipo
Squirtle	Água
Magikarp	Água
Totodile	Água

> `AND` e `OR` **(operadores lógicos)**: utilizamos em casos onde queremos adicionar mais de uma condição ou filtro no comando de consulta

O `AND` é utilizado quando precisamos de dados que atendam todas as condições estabelecidas. Por exemplo, queremos que retorne todos os pokemons que são do tipo água e tem hp > 45:

```
SELECT * FROM pokemons WHERE Tipo = 'Água' AND HP > 45;
```

Número	Nome	Tipo	Evolução	HP
158	Totodile	Água	Croconaw	50

Veja que, embora **4 Pokémons** atendam ao critério de ser do tipo **Água**, nem todos eles satisfazem o segundo critério, que exige um **HP maior que 45**.

O `OR` é utilizando quando precisamos que de dados que atendam pelo menos uma das condições estabelecidas. Por exemplo, queremos que retorne os pokemons do tipo életrico ou que possuem HP igual a 100.

```
SELECT * FROM pokemons WHERE Tipo = 'Életrico' OR HP = 100;
```

Número	Nome	Tipo	Evolução	HP
025	Pikachu	Elétrico	Raichu	35
151	Mew	Psíquico	-	100

Observe que as duas linhas retornadas atendem a apenas **um dos critérios**, e não necessariamente a ambos. Esse é exatamente o objetivo do operador `OR`: ele permite que a consulta retorne registros que satisfaçam **pelo menos uma** das condições especificadas, sem a necessidade de atender a todas as condições simultaneamente.

Operadores Lógicos e Condicionais

6. **Operadores Condicionais**: São usados para comparar valores e filtrar resultados com base em condições específicas.
7. **Operadores Lógicos**: Combinam múltiplas condições em uma única consulta.

Tipo de Operador	Operador	Descrição	Exemplo de Uso
Condicionais	`=`	Igual a	`SELECT * FROM pokemons WHERE Tipo = 'Água';`
	`≠` ou `<>`	Diferente de	`SELECT * FROM pokemons WHERE Tipo ≠ 'Fogo';`
	`>`	Maior que	`SELECT * FROM pokemons WHERE HP > 50;`
	`<`	Menor que	`SELECT * FROM pokemons WHERE HP < 30;`
	`≥`	Maior ou igual a	`SELECT * FROM pokemons WHERE HP ≥ 40;`
	`≤`	Menor ou igual a	`SELECT * FROM pokemons WHERE HP ≤ 60;`
	`BETWEEN`	Entre um intervalo (inclusivo)	`SELECT * FROM pokemons WHERE HP BETWEEN 30 AND 50;`
	`LIKE`	Busca por padrões (usando `%` para qualquer caractere e `_` para um único)	`SELECT * FROM pokemons WHERE Nome LIKE 'Pik%';`
	`IN`	Verifica se um valor está em uma lista	`SELECT * FROM pokemons WHERE Tipo IN ('Água', 'Fogo');`
	`IS NULL`	Verifica se um valor é nulo	`SELECT * FROM pokemons WHERE Evolução IS NULL;`
	`IS NOT NULL`	Verifica se um valor não é nulo	`SELECT * FROM pokemons WHERE Evolução IS NOT NULL;`
Lógicos	`AND`	Retorna verdadeiro se **todas** as condições forem verdadeiras	`SELECT * FROM pokemons WHERE Tipo = 'Água' AND HP > 40;`
	`OR`	Retorna verdadeiro se **pelo menos uma** condição for verdadeira	`SELECT * FROM pokemons WHERE Tipo = 'Água' OR HP > 50;`
	`NOT`	Inverte o resultado de uma condição	`SELECT * FROM pokemons WHERE NOT Tipo = 'Fogo';`

Banco de Dados do Mario

```sql
-- \c conectando no banco
CONNECT mario_database;

-- Criando uma banco de dados chamada mario_database a partir de outra (alterando o nome)
ALTER DATABASE first_database RENAME TO mario_database;
CREATE TABLE characters();

-- tipo de dado int que se auto incrementa a cada novo registro
ALTER TABLE characters ADD COLUMN character_id SERIAL;
ALTER TABLE characters ADD COLUMN name VARCHAR(30) NOT NULL;
ALTER TABLE characters ADD COLUMN homeland VARCHAR(60);
ALTER TABLE characters ADD COLUMN favorite_color VARCHAR(30);

-- \d characters
DESCRIBE characters;

-- Inserindo dados na tabela
INSERT INTO characters(name, homeland, favorite_color) VALUES
('Mario', 'Mushroom Kingdom', 'Red'),
('Luigi', 'Mushroom Kingdom', 'Green'),
('Peach', 'Mushroom Kingdom', 'Pink'),
('Toadstool', 'Mushroom Kingdom', 'Red'),
('Bowser', 'Mushroom Kingdom', 'Green'),
('Daisy', 'Sarasaland', 'Yellow'),
('Yoshi', 'Dinosaur Land', 'Green');

SELECT * FROM characters;
```

```sql
-- Alterando um registro da tabela:
UPDATE characters SET favorite_color = 'Orange' WHERE name = 'Daisy';
UPDATE characters SET name = 'Toad' WHERE favorite_color = 'Red';
UPDATE characters SET name = 'Mario' WHERE character_id = '1';
UPDATE characters SET favorite_color = 'Blue' WHERE name = 'Toad';
UPDATE characters SET favorite_color = 'Yellow' WHERE name = 'Bowser';
UPDATE characters SET homeland = 'Koopa Kingdom' WHERE name = 'Bowser';

SELECT * FROM characters ORDER BY character_id;

-- Transformando a coluna name como chave primária da tabela;
ALTER TABLE characters ADD PRIMARY KEY(name);

-- Retirando a restrição PRIMARY KEY da coluna name
ALTER TABLE characters DROP CONSTRAINT characters_pkey;

-- Coluna character_id será a PK
ALTER TABLE characters ADD PRIMARY KEY(character_id);

CREATE TABLE more_info();

ALTER TABLE more_info ADD COLUMN more_info_id SERIAL;
ALTER TABLE more_info ADD PRIMARY KEY(more_info_id);
ALTER TABLE more_info ADD COLUMN birthday DATE;
ALTER TABLE more_info ADD COLUMN height INT;
ALTER TABLE more_info ADD COLUMN weight NUMERIC(4, 1);

--\d more_info
DESCRIBE more_info;

-- adicionando uma chave estrangeira na tabela more_info
ALTER TABLE more_info ADD COLUMN character_id INT REFERENCES characters(character_id);

-- Adicionando uma restrição de valores unicos na chave estrangeira
ALTER TABLE more_info ADD UNIQUE(character_id);

-- Adicionando outra restrição que não permite valores nulos para a FK
ALTER TABLE more_info ALTER COLUMN character_id SET NOT NULL;

SELECT character_id, name FROM characters;

INSERT INTO more_info(character_id, birthday, height, weight) VALUES
(1, '1981-07-09', 155, 64.5),
(2, '1983-07-14', 175, 48.8),
(3, '1985-10-18', 173, 52.2),
(4, '1950-01-10', 66, 35.6),
(5, '1990-10-29', 258, 300),
(6, '1989-07-31', NULL, NULL),
(7, '1990-04-13 ', 162, 59.1);


SELECT * FROM more_info;
SELECT character_id, name FROM characters WHERE name = 'Toad';

-- Nome do brouser tava errado
UPDATE characters SET name = 'Browser' WHERE character_id = 5;
SELECT character_id, name FROM characters WHERE name = 'Bowser';
SELECT character_id, name FROM characters WHERE name = 'Daisy';
SELECT character_id, name FROM characters WHERE name = 'Yoshi';



ALTER TABLE more_info RENAME COLUMN height TO height_in_cm;
ALTER TABLE more_info RENAME COLUMN weight TO weight_in_kg;

-- criando uma tabela e já adicionando uma coluna, em vez de alterar ela toda hora
CREATE TABLE sounds(sound_id SERIAL PRIMARY KEY);

ALTER TABLE sounds ADD COLUMN filename VARCHAR(40) NOT NULL UNIQUE;

-- aqui estamos definindo a coluna character_is da tabela de characters como chave estrangeira na tabela sounds
-- será um relacionamento 1 para muitos, ou seja, um character pode ter vários sons, mas um som só pode ter um character
associado a ele.

-- ALTER TABLE table_name ADD COLUMN column_name DATATYPE CONSTRAINT REFERENCES
referenced_table_name(referenced_column_name);
ALTER TABLE sounds ADD COLUMN character_id INT NOT NULL REFERENCES characters(character_id);

SELECT * FROM characters ORDER BY character_id;
```

```sql
INSERT INTO sounds(filename, character_id) VALUES
('its-a-me.wav', 1),
('yippee.wav', 1),
('ha-ha.wav', 2),
('oh-yeah.wav', 2),
('woo-hoo.wav', 3),
('yay.wav', 3),
('mm-hmm.wav', 3),
('yahoo.wav', 1);

SELECT * FROM sounds;

-- Nessa tabela, não será necessário uma chave estrangeira, pois mais de um character pode executar mais de uma ação, então será uma relação muitos-para-muitos
CREATE TABLE actions(action_id SERIAL PRIMARY KEY);

ALTER TABLE actions ADD COLUMN action VARCHAR(20) UNIQUE NOT NULL;

INSERT INTO actions(action) VALUES('run');
INSERT INTO actions(action) VALUES('jump');
INSERT INTO actions(action) VALUES('duck');

CREATE TABLE character_actions();

ALTER TABLE character_actions ADD COLUMN character_id INT NOT NULL REFERENCES characters(character_id);
ALTER TABLE character_actions ADD COLUMN action_id INT NOT NULL REFERENCES actions(action_id);
ALTER TABLE character_actions ADD PRIMARY KEY(character_id, action_id);

INSERT INTO character_actions(character_id, action_id) VALUES
(1, 1),
(1, 2),
(1, 3);

select * from character_actions;

-- SELECT columns FROM table_1 FULL JOIN table_2 ON table_1.primary_key_column = table_2.foreign_key_column;
SELECT * FROM characters c FULL JOIN more_info m ON c.character_id = m.character_id;
SELECT * FROM characters c FULL JOIN sounds s ON c.character_id = s.character_id;

-- SELECT columns FROM junction_table
-- FULL JOIN table_1 ON junction_table.foreign_key_column = table_1.primary_key_column
-- FULL JOIN table_2 ON junction_table.foreign_key_column = table_2.primary_key_column;
SELECT * FROM character_actions
FULL JOIN actions ON character_actions.action_id = actions.action_id
FULL JOIN characters ON characters.character_id = character_actions.character_id;
```