

Banco de  
Dados  
Com MySQL



WOMAKERS<sup>®</sup>  
CODE

## Silas Lira

Especialista de Software III @ E-Core

<https://womakerscode.org>

<https://www.e-core.com>

<https://www.linkedin.com/in/silaslira/>

---



# Agenda

## Aula 1 - 24/01/2023

- O que é um banco de dados
- Como instalar o MySQL
- Consultas com `SELECT`
- Filtrar uma consulta com `WHERE`
- Ordenar uma consulta com `ORDER BY`

## Aula 2 - 25/01/2023

- Adicionar dados com `INSERT`
- Atualizar dados com `UPDATE`
- Remover dados com `DELETE`
- Criar tabelas com `CREATE TABLE`
- Alterar tabelas com `ALTER TABLE`

# Agenda

## Aula 3 - 26/01/2023

- Pagar uma consulta com **LIMIT**
- Filtrar uma consulta com **HAVING**
- Fazer a união entre tabelas para uma consulta com **JOIN**

## Aula 4 - 31/01/2023

- Utilizar Sub-Queries
- Funções do MySQL
- Agrupar dados com **GROUP BY**
- Dúvidas e Exercícios

# INSERT, UPDATE, DELETE e Transações

Abrindo uma transação que altera dados em uma tabela (DML)

Existe um grupo de instruções chamado **Data Manipulation Language - DML** (Linguagem de Manipulação de Dados). São instruções responsáveis por manipular dados através da inserção (**INSERT**), atualização (**UPDATE**) e da remoção (**DELETE**).

Estes comandos SQL são executados em uma operação chamada **Transaction** (transação)

Também existe um conjunto de instruções responsável por controlar uma transação, conhecidos como **Transaction Control Language - TCL** (Linguagem de Controle de Transação). Esse conjunto contém duas instruções muito importantes que são o **COMMIT** e o **ROLLBACK**.

Especificamente, o **SELECT** é o único que não precisa de um controle explícito.

O **MySQL Workbench** trabalha com uma opção de controle de transações automático.

# INSERT, UPDATE, DELETE e Transações

Abrindo uma transação que altera dados em uma tabela (DML)

Esses comandos são executados em uma operação chamada **Transaction** (transação)

Também existe um conjunto de instruções responsável por controlar uma transação, conhecidos como **Transaction Control Language - TCL** (Linguagem de Controle de Transação). Esse conjunto contém duas instruções muito importantes que são o **COMMIT** e o **ROLLBACK**

O **MySQL Workbench** trabalha com uma opção de controle de transações automático. Vamos explorar um pouco como trabalhar com ele ativado e desativado.

# COMMIT e ROLLBACK

## Controlando uma transação

O **COMMIT** é um comando que “aplica” as modificações feitas durante uma transação.

Ao modificar um dado com um comando do tipo **DML**, essas alterações não ficam salvas até a execução de um **COMMIT**

Isso nos permite validar se os dados novos são realmente aquilo que esperamos, e que nada foi afetado de forma não intencional

Caso alguma alteração tenha sido feita por engano podemos utilizar o comando **ROLLBACK**.

Este comando desfaz as alterações que estão pendentes na transação, sem afetar o banco de dados.

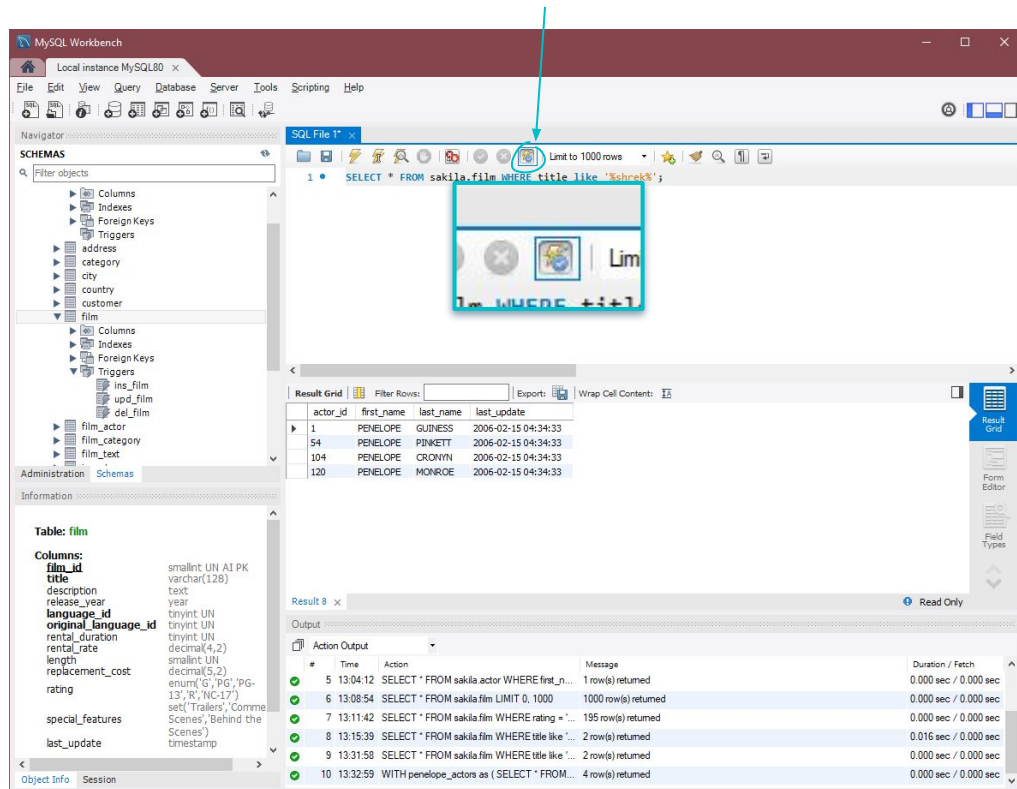
# COMMIT e ROLLBACK

## Controlando uma transação

O controle automático de transação do **MySQL Workbench** adiciona de forma implícita um **COMMIT** em cada uma das nossas transações.

Podemos observar na parte superior da janela o botão que liga/desliga essa opção.

Vamos manter essa opção ligada por enquanto.





# INSERT

Inserindo dados em uma tabela

Para inserir dados em uma tabela vamos utilizar o verbo **"INSERT INTO"** (INSIRA EM)

A estrutura básica do comando é a seguinte: **INSERT INTO** [schema.tabela] (coluna1, coluna2, ... colunaX) **VALUES** (valor1, valor2, ..., valorX);

\*Essa estrutura pode ser traduzida como: **INSIRA EM** [schema.tabela] (coluna1, coluna2, ... colunaX) **VALORES** (valor1, valor2, ..., valorX);

Os grupos de colunas e valores devem estar encapsulados em listas definidas por parêntesis **"()**

Quando executamos um insert devemos explicitar as colunas que serão inseridas, pois podem existir colunas que possuem dados gerados automaticamente, ou podemos querer deixar uma coluna propositalmente vazia

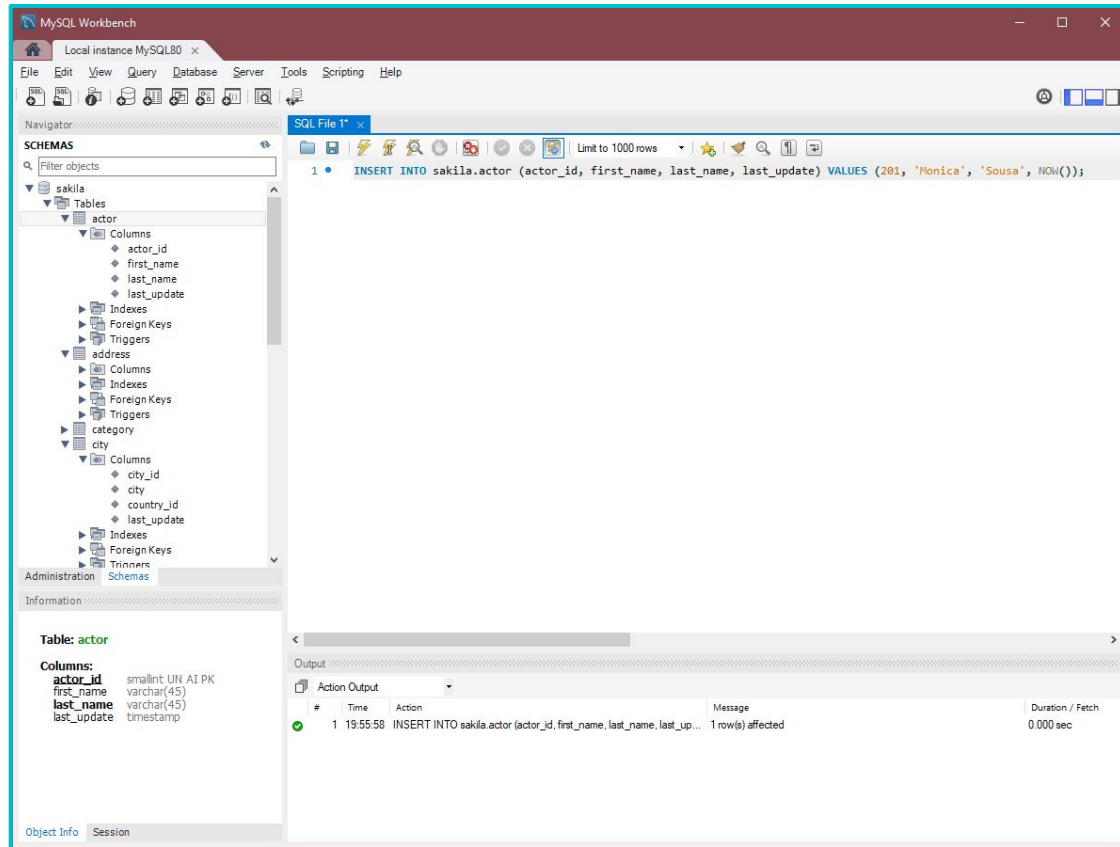
Os valores são os valores que devem ser inseridos em cada coluna. A ordem dos valores irá respeitar a ordem das colunas definidas. O primeiro valor vai para a primeira coluna definida, o segundo para a segunda e assim por diante

# INSERT

Inserindo dados em uma tabela

Vamos adicionar um novo ator na nossa tabela.

**"INSERT INTO sakila.actor  
(actor\_id, first\_name, last\_name,  
last\_update) VALUES (201,  
'Monica', 'Sousa', NOW());"**



The screenshot displays the MySQL Workbench interface. On the left, the 'SCHEMAS' pane shows the 'sakila' database expanded, with the 'actor' table selected. The 'Columns' pane for the 'actor' table lists: actor\_id (smallint, UNSIGNED, PRIMARY KEY), first\_name (varchar(45)), last\_name (varchar(45)), and last\_update (timestamp). The main editor shows the SQL statement: `INSERT INTO sakila.actor (actor_id, first_name, last_name, last_update) VALUES (201, 'Monica', 'Sousa', NOW());`. The 'Output' pane at the bottom shows the execution results:

#	Time	Action	Message	Duration / Fetch
1	19:55:58	INSERT INTO sakila.actor (actor_id, first_name, last_name, last_update)	1 row(s) affected	0.000 sec

# INSERT

Inserindo dados em uma tabela

Explicando a query:

Inserimos na tabela sakila.actor os seguintes dados:

actor\_id: 201

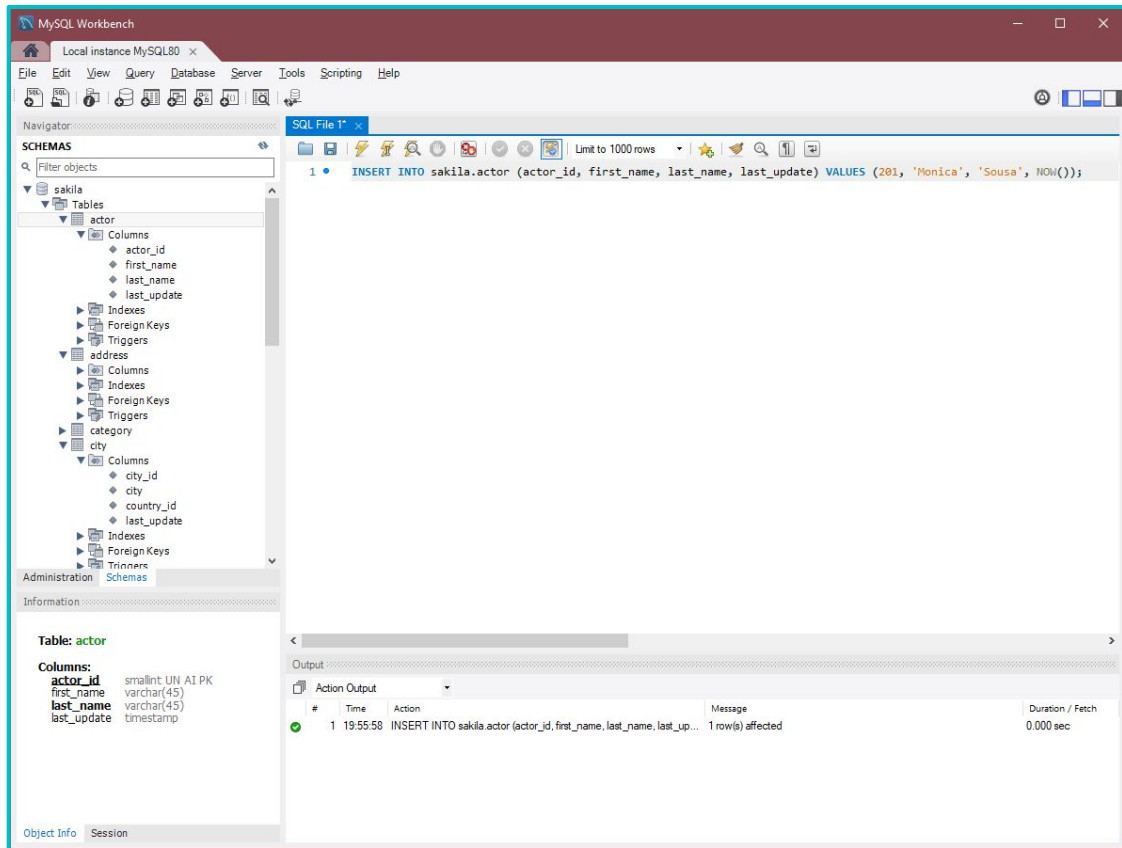
first\_name: Monica

last\_name: Sousa

last\_update: **NOW()**

A Id nós podemos descobrir consultando a tabela **actor**, verificando a maior id e definindo como o próximo valor

A função **NOW()** (AGORA()) gera um valor do tipo **TimeStamp** com o momento atual do relógio do sistema com precisão de segundos.



# INSERT

Inserindo dados em uma tabela

Vamos tentar inserir somente alguns dados e ver como o MySQL se comporta?

“**INSERT INTO** sakila.actor (first\_name, last\_name) **VALUES** ('Cebolinha', 'Sousa');”

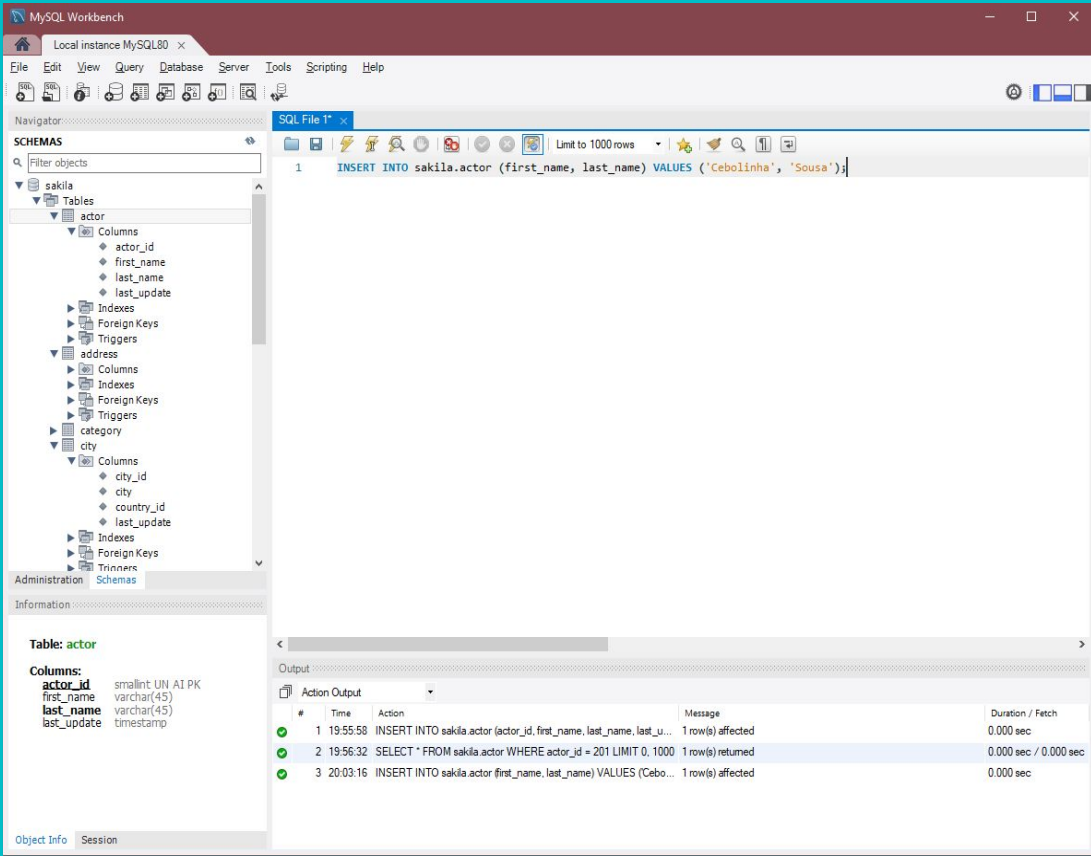
The screenshot shows the MySQL Workbench interface. On the left, the 'SCHEMAS' pane displays the 'sakila' database structure, including tables like 'actor', 'address', 'category', and 'city'. The 'actor' table is selected, showing its columns: 'actor\_id' (smallint UNSIGNED, AI, PK), 'first\_name' (varchar(45)), 'last\_name' (varchar(45)), and 'last\_update' (timestamp). The 'Output' pane at the bottom shows the execution of the SQL statement: `INSERT INTO sakila.actor (first_name, last_name) VALUES ('Cebolinha', 'Sousa');`. The output table has columns: #, Time, Action, Message, and Duration / Fetch. The message indicates that 1 row(s) were affected.

#	Time	Action	Message	Duration / Fetch
1	19:55:58	INSERT INTO sakila.actor (actor_id, first_name, last_name, last_u...	1 row(s) affected	0.000 sec
2	19:56:32	SELECT * FROM sakila.actor WHERE actor_id = 201 LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
3	20:03:16	INSERT INTO sakila.actor (first_name, last_name) VALUES ('Cebol...	1 row(s) affected	0.000 sec

# INSERT

Inserindo dados em uma tabela

Esse comando funciona pois existem definições na própria tabela que permitem a geração automática dos valores actor\_id e last\_update. Vamos explorar esses atributos com mais detalhes em breve



The screenshot displays the MySQL Workbench interface. On the left, the 'SCHEMAS' pane shows the 'sakila' database structure, including tables like 'actor' and 'address'. The 'actor' table is selected, showing its columns: 'actor\_id', 'first\_name', 'last\_name', and 'last\_update'. Below this, the 'Table: actor' details are shown, including data types and constraints.

The main SQL editor shows the following SQL statement:

```
1 INSERT INTO sakila.actor (first_name, last_name) VALUES ('CeboLinha', 'Sousa');
```

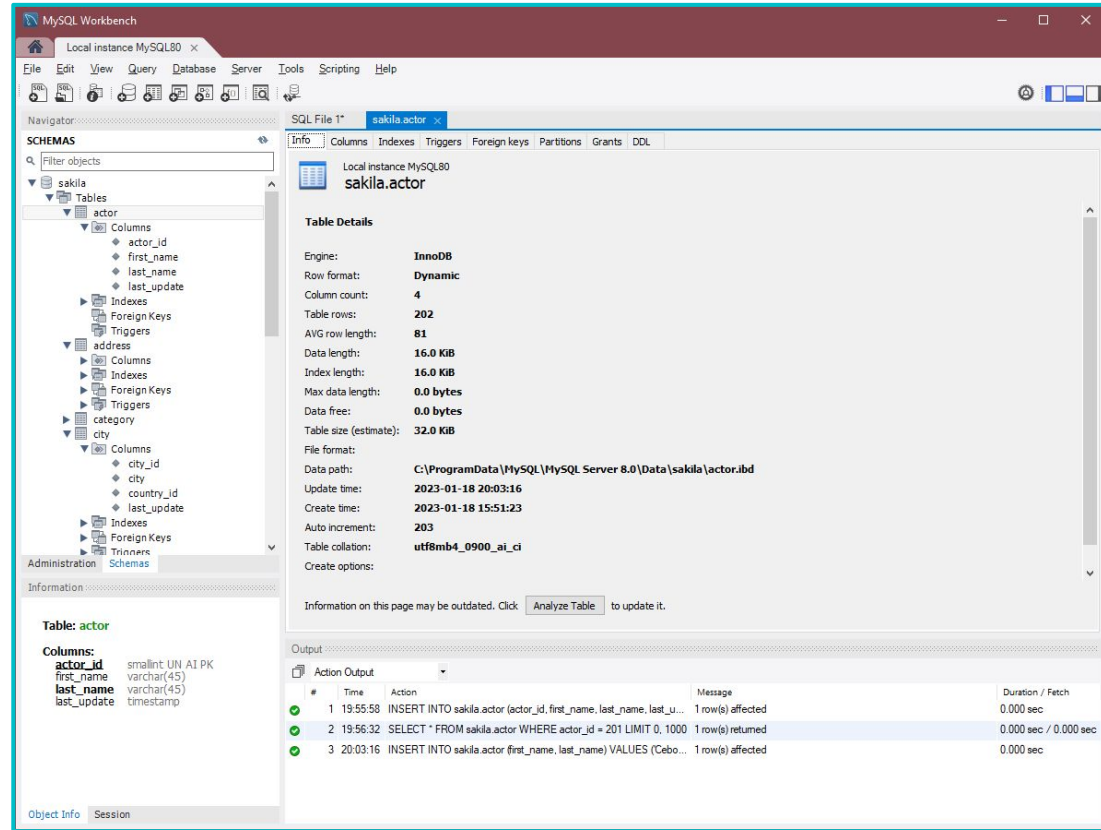
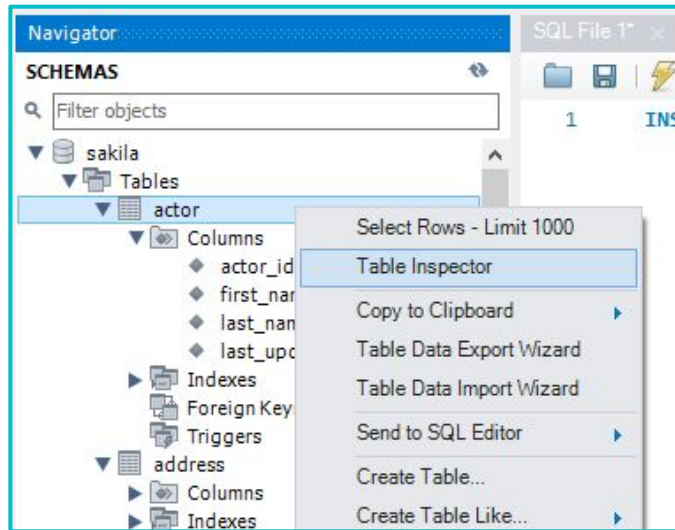
The 'Output' pane at the bottom shows the execution results of the SQL statement. It includes a table with columns: #, Time, Action, Message, and Duration / Fetch.

#	Time	Action	Message	Duration / Fetch
1	19:55:58	INSERT INTO sakila.actor (actor_id, first_name, last_name, last_u...	1 row(s) affected	0.000 sec
2	19:56:32	SELECT * FROM sakila.actor WHERE actor_id = 201 LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
3	20:03:16	INSERT INTO sakila.actor (first_name, last_name) VALUES (Cebo...	1 row(s) affected	0.000 sec

# INSERT

Inserindo dados em uma tabela

Clicando com o botão direito na tabela **actor** e selecionando **Table inspector** podemos ver as **definições da tabela**



# INSERT

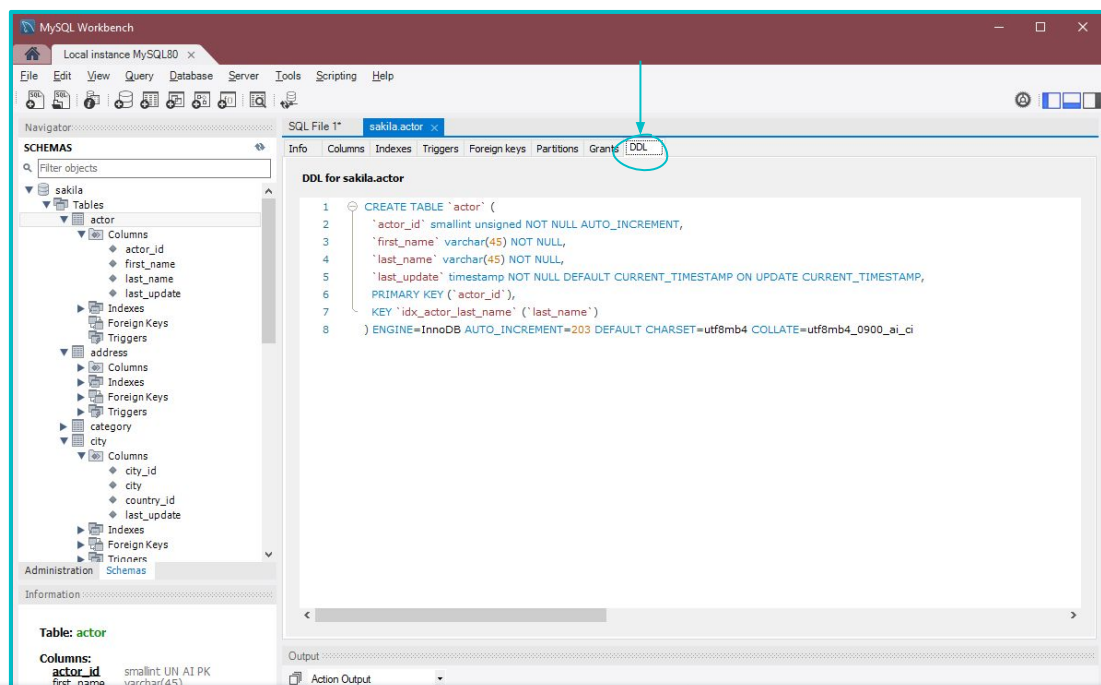
Inserindo dados em uma tabela

Selecionando a aba **DDL** podemos ver a definição da tabela **actor**

Note que na definição do campo actor id existe uma regra **AUTO\_INCREMENT**  
Essa regra define que a id é incrementada **automaticamente**

Na definição de last update existe também uma definição que diz **DEFAULT**  
Ou seja, um valor padrão.

Essas são as únicas informações que importam por agora.



```
1 CREATE TABLE `actor` (  
2   `actor_id` smallint unsigned NOT NULL AUTO_INCREMENT,  
3   `first_name` varchar(45) NOT NULL,  
4   `last_name` varchar(45) NOT NULL,  
5   `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  
6   PRIMARY KEY (`actor_id`),  
7   KEY `idx_actor_last_name` (`last_name`)  
8 ) ENGINE=InnoDB AUTO_INCREMENT=203 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

# INSERT

## Exercícios

1- Insira um novo ator na tabela **actor** passando **MANUALMENTE** todos os dados, incluindo **actor id** e **last update**

2- Insira um novo filme na tabela **film** passando os seguintes dados: **title** (título), **description** (descrição) e **language id** (Id do idioma)

\*Para obter uma **language id** consulte a tabela **language**

Desafio: Utilizando o **novo actor** e o **novo film** insira o **actor id** e o **film id** na tabela **film\_actor** para **relacionar o novo ator com o novo filme.**



# UPDATE

Atualizando dados em uma tabela

Para atualizar dados em uma tabela vamos utilizar o verbo “**UPDATE**” (ATUALIZE)

A estrutura básica do comando é a seguinte: **UPDATE** [schema.tabela] **SET** (coluna1 = valor, coluna2 = valor, colunaN = valor) **WHERE** [condição];

\*Essa estrutura pode ser traduzida como: **ATUALIZE** [schema.tabela] **DEFINA** (coluna1 = valor, coluna2 = valor, colunaN = valor) **ONDE** [condição];

Neste comando definimos qual coluna queremos atualizar e qual será o novo valor.

No **UPDATE** é muito importante definir a condição **WHERE**. Sem ela, a atualização será feita na **TABELA INTEIRA**. Por exemplo: se eu atualizar um nome sem definir o **WHERE**, **todos os nomes da tabela** serão definidos como esse novo valor.

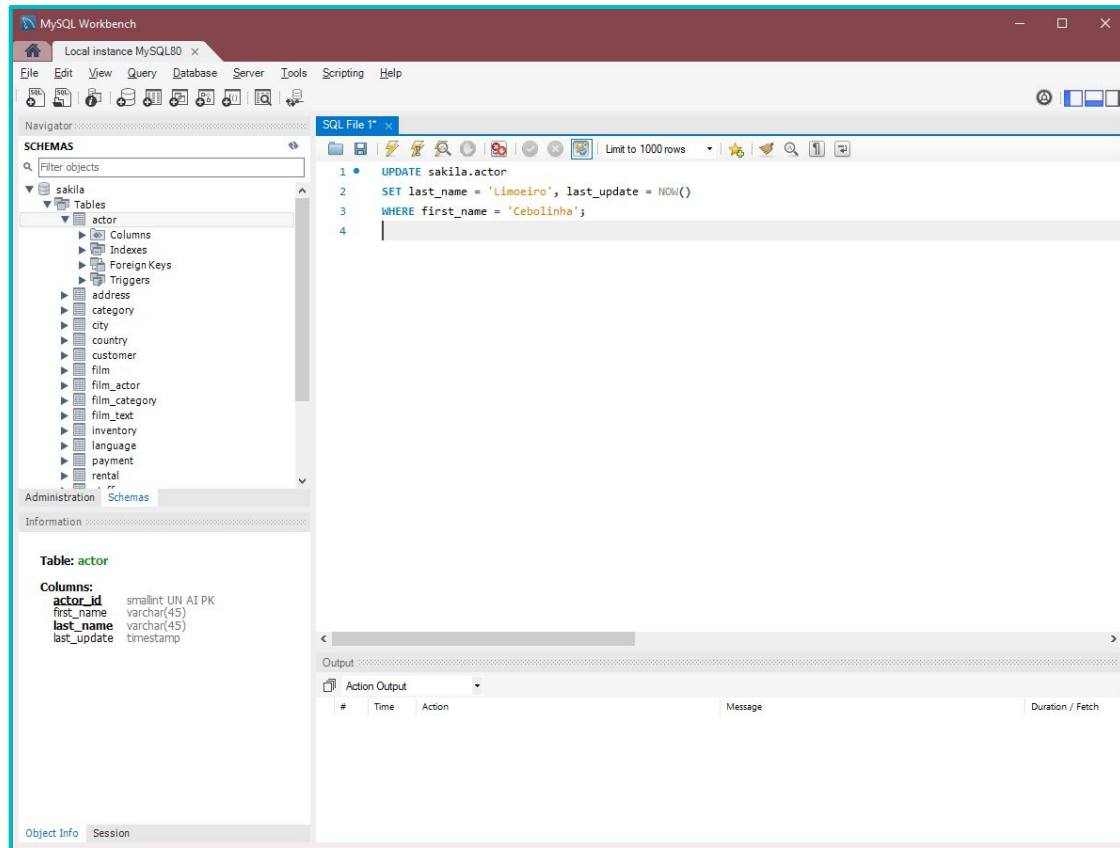
A condição **WHERE** geralmente é definida com base na ID do registro que queremos atualizar, mas pode ser qualquer valor, igual em um **SELECT**. Inclusive, a condição pode ser o resultado de um **SELECT** e vamos explorar essa opção quando falarmos de **Sub-Queries**

# UPDATE

Atualizando dados em uma tabela

Vamos alterar um dos nossos registros

```
"UPDATE sakila.actor  
SET last_name = 'Limoeiro',  
last_update = NOW()  
WHERE first_name = 'Cebolinha';"
```



# UPDATE

## Atualizando dados em uma tabela

Explicando a query:

Estamos atualizando o último nome de um ator

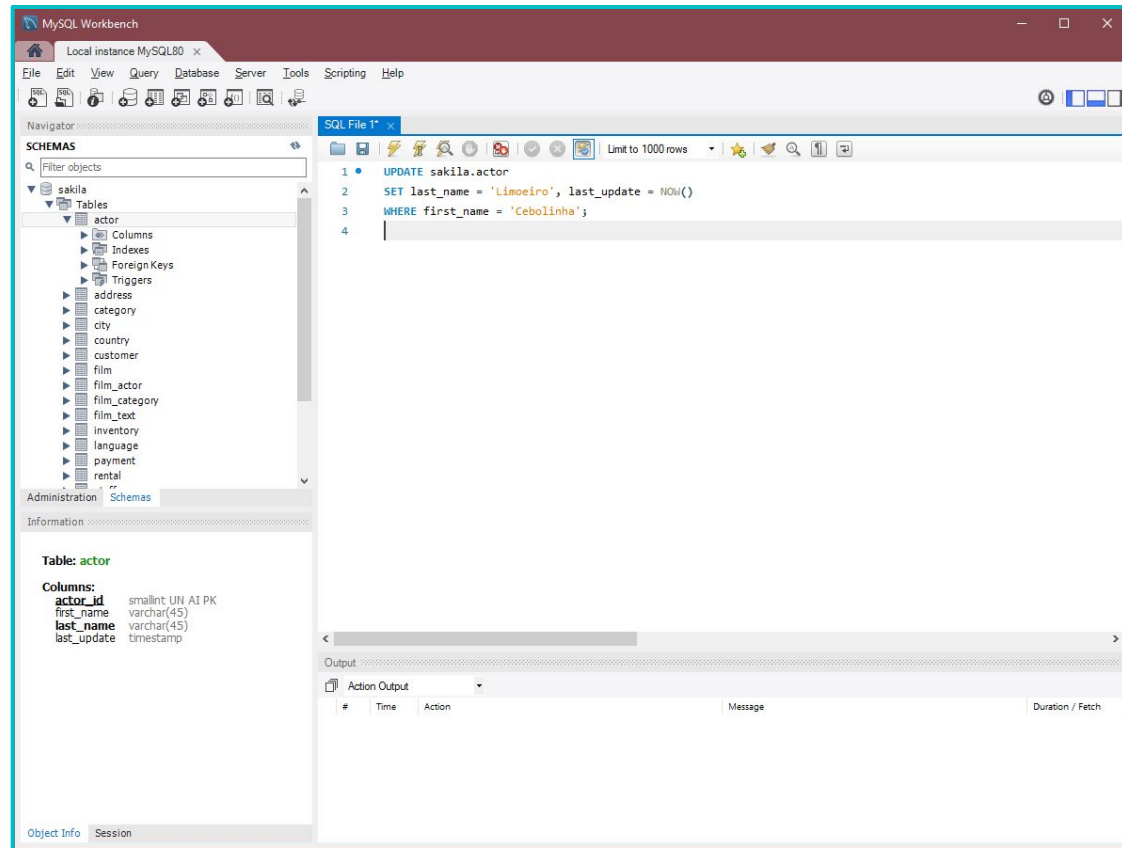
**"SET** last\_name = 'Limoeiro'  
Last\_update = NOW()"

Estamos dizendo que na tabela sakila.actor o atributo last\_name deve ser igual à "Limoeiro"

A data de última atualização é "agora".  
Diferente do **INSERT** não usaremos o valor "padrão" então precisamos passar um valor novo

Definimos no **WHERE** que desejamos apenas atualizar o registro onde o primeiro nome seja igual a "Cebolinha".

Sem isso, TODOS os registros seriam alterados.



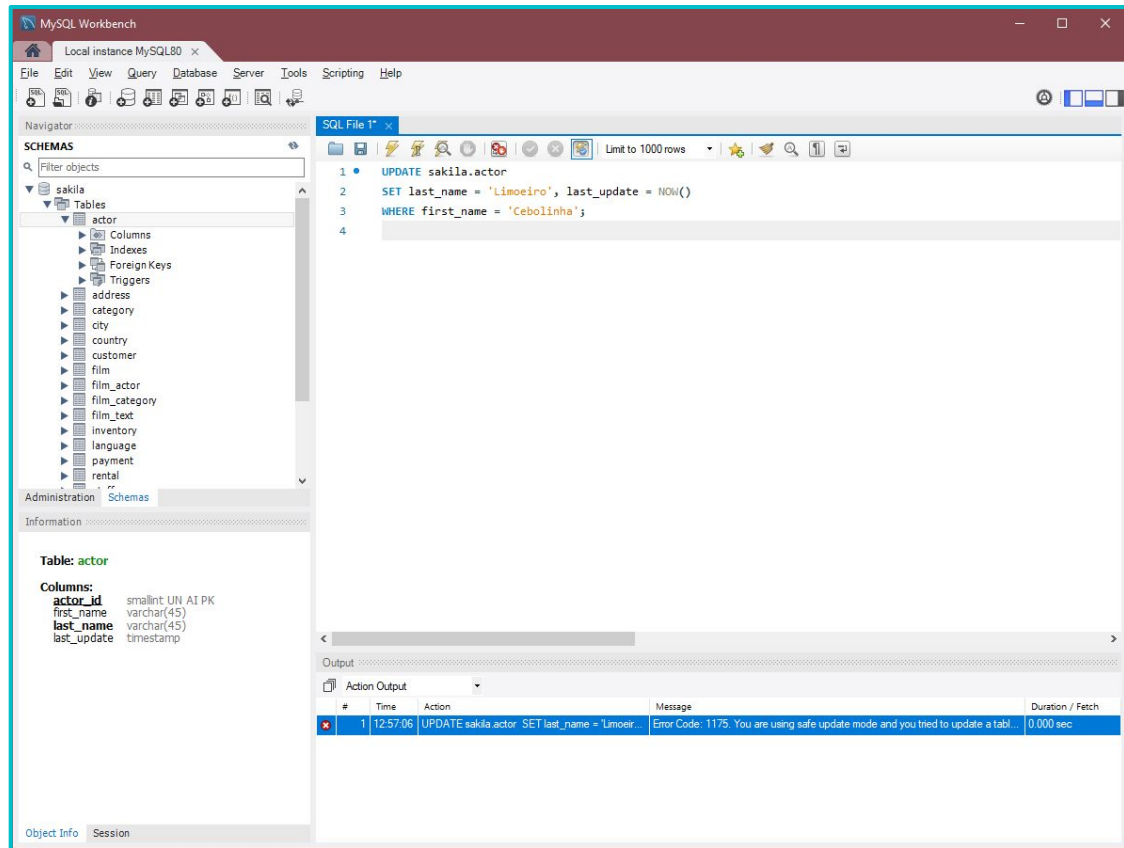
# UPDATE

Atualizando dados em uma tabela

Ao tentar executar essa consulta, podemos ver que o MySQL trará um **erro no nosso console de ações**

Esse erro acontece pois estamos usando o modo “seguro” do SQL e ele não permite executar um comando **UPDATE** onde o **WHERE** não faça referência à uma chave (PK).

Podemos alterar essa configuração, mas por enquanto vamos alterar a nossa consulta.



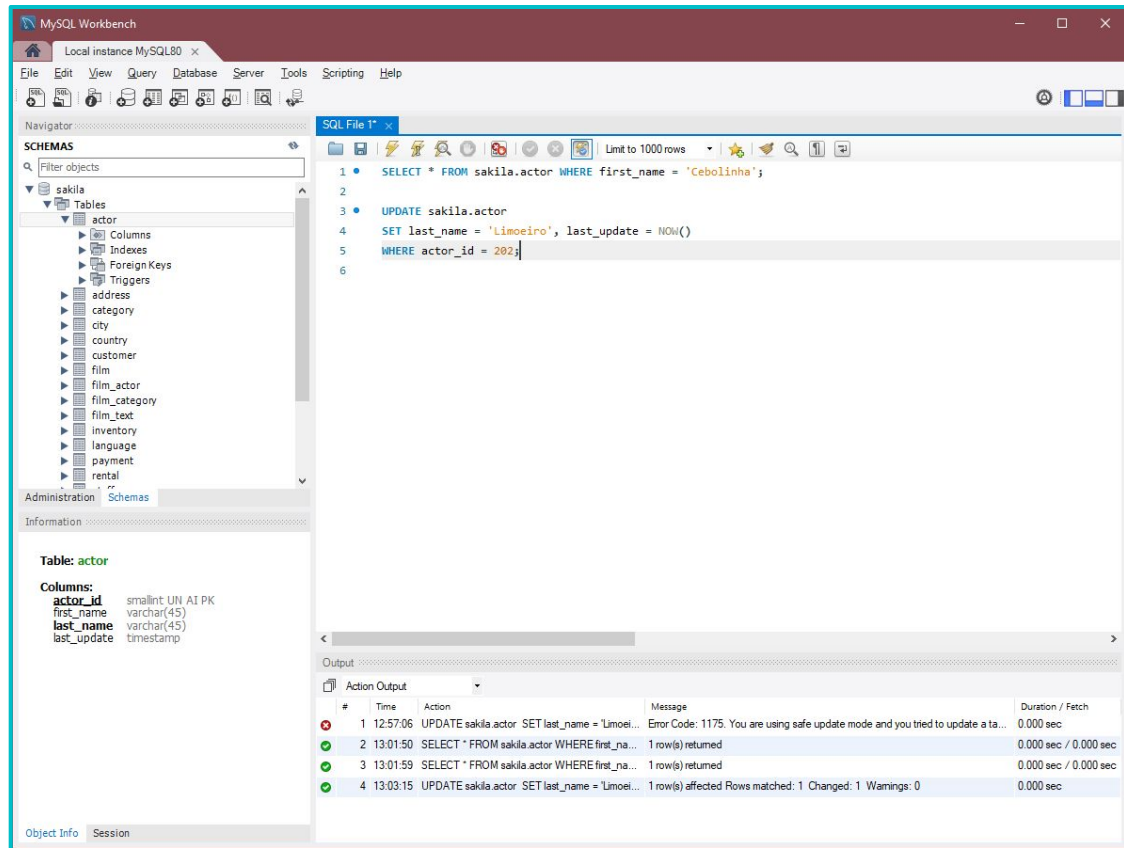
# UPDATE

Atualizando dados em uma tabela

Vamos executar um **SELECT** para descobriremos qual a actor\_id que queremos atualizar.

E vamos alterar o nosso **UPDATE** para ter como referência o actor\_id = 202

Ao executar a query, podemos ver no console de ações que 1 row (linha) foi afetada



The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'Schemas' tree with 'sakila' expanded, showing tables like 'actor', 'address', 'category', etc. The main editor shows a SQL script with two queries: a SELECT and an UPDATE. The output pane at the bottom shows the execution results.

```
1 • SELECT * FROM sakila.actor WHERE first_name = 'CeboLinha';
2
3 • UPDATE sakila.actor
4   SET last_name = 'Limoeiro', last_update = NOW()
5   WHERE actor_id = 202;
```

Output:

#	Time	Action	Message	Duration / Fetch
1	12:57:06	UPDATE sakila.actor SET last_name = 'Limoei...	Error Code: 1175. You are using safe update mode and you tried to update a ta...	0.000 sec
2	13:01:50	SELECT * FROM sakila.actor WHERE first_na...	1 row(s) returned	0.000 sec / 0.000 sec
3	13:01:59	SELECT * FROM sakila.actor WHERE first_na...	1 row(s) returned	0.000 sec / 0.000 sec
4	13:03:15	UPDATE sakila.actor SET last_name = 'Limoei...	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0	0.000 sec

# UPDATE

Atualizando dados em uma tabela

Agora, ao consultar a nossa tabela de atores, podemos ver que o sobrenome do Cebolinha foi atualizado.

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'Schemas' tree with 'sakila' expanded, showing tables like 'actor', 'address', 'category', etc. The main editor window shows a SQL query: `SELECT * FROM sakila.actor WHERE first_name = 'Cebolinha';`. Below the query, the 'Result Grid' shows one row: actor\_id 202, first\_name Cebolinha, last\_name Limoeiro, last\_update 2023-01-20 13:03:15. The bottom panel shows the 'Output' tab with a table of actions and their results.

#	Time	Action	Message	Duration / Fetch
1	12:57:06	UPDATE sakila.actor SET last_name = 'Limoei...	Error Code: 1175. You are using safe update mode and you tried to update a ta...	0.000 sec
2	13:01:50	SELECT * FROM sakila.actor WHERE first_na...	1 row(s) returned	0.000 sec / 0.000 sec
3	13:01:59	SELECT * FROM sakila.actor WHERE first_na...	1 row(s) returned	0.000 sec / 0.000 sec
4	13:03:15	UPDATE sakila.actor SET last_name = 'Limoei...	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0	0.000 sec
5	13:04:12	SELECT * FROM sakila.actor WHERE first_na...	1 row(s) returned	0.000 sec / 0.000 sec

# UPDATE

## Exercícios

1- Atualize na tabela **film** o valor do atributo rental\_rate para 3.99 para o filme de title "ALONE TRIP". Passando um valor para last\_update

2- Atualize na tabela **film** o valor dos atributos rental\_duration para 10, replacement\_cost para 5.00 do filme "Citizen Shrek"

# DELETE

## Removendo dados em uma tabela

Para remover dados em uma tabela vamos utilizar o verbo “**DELETE**” (APAGUE)

A estrutura básica do comando é a seguinte: **DELETE FROM** [schema.tabela] **WHERE** [condição];

\*Essa estrutura pode ser traduzida como: **APAGUE DE** [schema.tabela] **ONDE** [condição];

Neste comando, indicamos a tabela onde desejamos remover uma linha.

No **DELETE** é muito importante definir a condição **WHERE**. Sem ela, a remoção será feita na **TABELA INTEIRA**

Por exemplo: se eu deletar sem definir o **WHERE**, **todos os dados da tabela serão apagados**.

A condição **WHERE** geralmente é definida com base na ID do registro que queremos atualizar, mas pode ser qualquer valor, igual em um **SELECT**. Inclusive, a condição pode ser o resultado de um **SELECT** e vamos explorar essa opção quando falarmos de **Sub-Queries**



# DELETE

Removendo dados em uma tabela

Vamos primeiro inserir um novo dado na tabela **actor**

**"INSERT INTO sakila.actor  
(first\_name, last\_name) VALUES  
('Teste', 'Testado');"**

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'sakila' database schema with tables like actor, address, category, city, country, customer, film, and film\_actor. The main editor window shows the SQL query: `INSERT INTO sakila.actor (first_name, last_name) VALUES ('Teste', 'Testado');`. The bottom output window shows the results of the query execution.

#	Time	Action	Message	Duration / Fetch
1	15:17:55	INSERT INTO sakila.actor first_name, last_na...	1 row(s) affected	0.000 sec
2	15:19:01	SELECT * FROM sakila.actor WHERE first_na...	2 row(s) returned	0.000 sec / 0.000 sec
3	15:19:24	DELETE FROM sakila.actor WHERE actor_id l...	2 row(s) affected	0.000 sec
4	15:19:27	SELECT * FROM sakila.actor WHERE first_na...	0 row(s) returned	0.000 sec / 0.000 sec
5	15:19:29	INSERT INTO sakila.actor first_name, last_na...	1 row(s) affected	0.000 sec
6	15:19:30	SELECT * FROM sakila.actor WHERE first_na...	1 row(s) returned	0.000 sec / 0.000 sec

# DELETE

Removendo dados em uma tabela

Vamos conferir se o nosso valor foi incluído corretamente.

**"SELECT \* FROM sakila.actor  
WHERE first\_name = 'Teste';"**

Verifique a ID do nosso actor. Nesse caso é 205, mas para você pode ser diferente dependendo de quantos registros você adicionou.

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'SCHEMAS' tree with 'sakila' expanded, showing tables like 'actor', 'address', 'category', 'city', 'country', 'customer', 'film', 'ins\_film', 'upd\_film', and 'del\_film'. The 'actor' table is selected, and its columns are listed: 'actor\_id' (smallint, UN, AI, PK), 'first\_name' (varchar(45)), 'last\_name' (varchar(45)), and 'last\_update' (timestamp).

The main window shows a SQL query in the 'SQL File 1\*' editor:

```
1 SELECT * FROM sakila.actor WHERE first_name = 'Teste';
```

The 'Result Grid' shows the query results:

actor_id	first_name	last_name	last_update
205	Teste	Testado	2023-01-20 15:19:29

The 'Output' tab shows the 'Action Output' log:

#	Time	Action	Message	Duration / Fetch
2	15:19:01	SELECT * FROM sakila.actor WHERE first_n...	2 row(s) returned	0.000 sec / 0.000 sec
3	15:19:24	DELETE FROM sakila.actor WHERE actor_id...	2 row(s) affected	0.000 sec
4	15:19:27	SELECT * FROM sakila.actor WHERE first_n...	0 row(s) returned	0.000 sec / 0.000 sec
5	15:19:29	INSERT INTO sakila.actor (first_name, last_n...	1 row(s) affected	0.000 sec
6	15:19:30	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec
7	15:21:16	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec

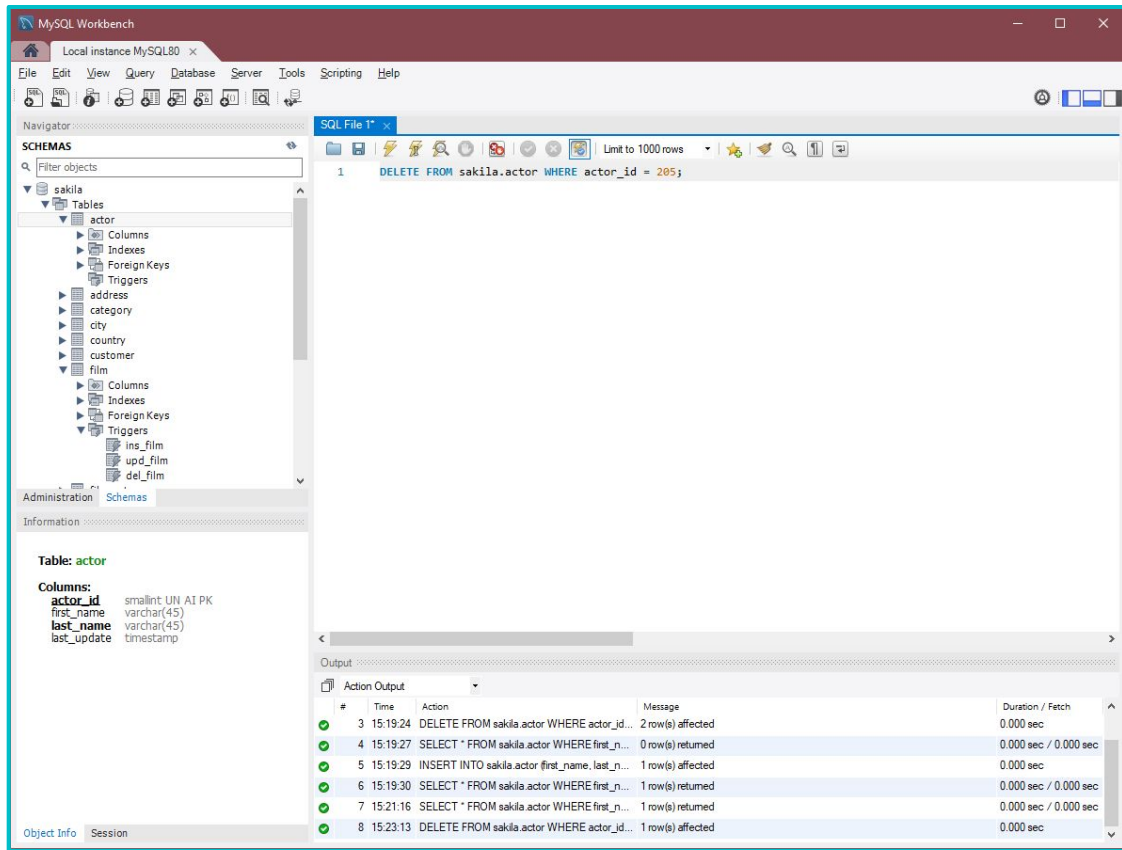
# DELETE

Removendo dados em uma tabela

Agora vamos deletar esse actor que acabamos de criar.

**"DELETE FROM sakila.actor WHERE actor\_id = 205;"**

No console de ações podemos ver que 1 linha foi afetada.



The screenshot shows the MySQL Workbench interface. The SQL Editor contains the query: `DELETE FROM sakila.actor WHERE actor_id = 205;`. The left sidebar shows the database structure with the `sakila` database selected. The `actor` table is highlighted. The bottom panel shows the 'Action Output' tab with the following results:

#	Time	Action	Message	Duration / Fetch
3	15:19:24	DELETE FROM sakila.actor WHERE actor_id...	2 row(s) affected	0.000 sec
4	15:19:27	SELECT * FROM sakila.actor WHERE first_n...	0 row(s) returned	0.000 sec / 0.000 sec
5	15:19:29	INSERT INTO sakila.actor (first_name, last_n...	1 row(s) affected	0.000 sec
6	15:19:30	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec
7	15:21:16	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec
8	15:23:13	DELETE FROM sakila.actor WHERE actor_id...	1 row(s) affected	0.000 sec

# DELETE

Removendo dados em uma tabela

Agora tentando selecionar novamente o actor "Teste".

Não encontramos nenhum resultado pois foi removido.

The screenshot shows the MySQL Workbench interface. On the left, the 'SCHEMAS' pane displays the 'sakila' database structure, including tables like 'actor', 'address', 'category', 'city', 'country', 'customer', 'film', 'ins\_film', 'upd\_film', and 'del\_film'. The 'actor' table is selected, showing its columns: 'actor\_id' (smallint, UN, AI, PK), 'first\_name' (varchar(45)), 'last\_name' (varchar(45)), and 'last\_update' (timestamp).

The main query editor shows the following SQL query:

```
SELECT * FROM sakila.actor WHERE first_name = 'Teste';
```

The 'Result Grid' pane shows the query results, which are empty, indicating that no records were found.

The 'Output' pane shows the execution log, including the following actions:

#	Time	Action	Message	Duration / Fetch
4	15:19:27	SELECT * FROM sakila.actor WHERE first_n...	0 row(s) returned	0.000 sec / 0.000 sec
5	15:19:29	INSERT INTO sakila.actor first_name, last_n...	1 row(s) affected	0.000 sec
6	15:19:30	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec
7	15:21:16	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec
8	15:23:13	DELETE FROM sakila.actor WHERE actor_id...	1 row(s) affected	0.000 sec
9	15:23:56	SELECT * FROM sakila.actor WHERE first_n...	0 row(s) returned	0.000 sec / 0.000 sec

# DELETE

## Exercicios

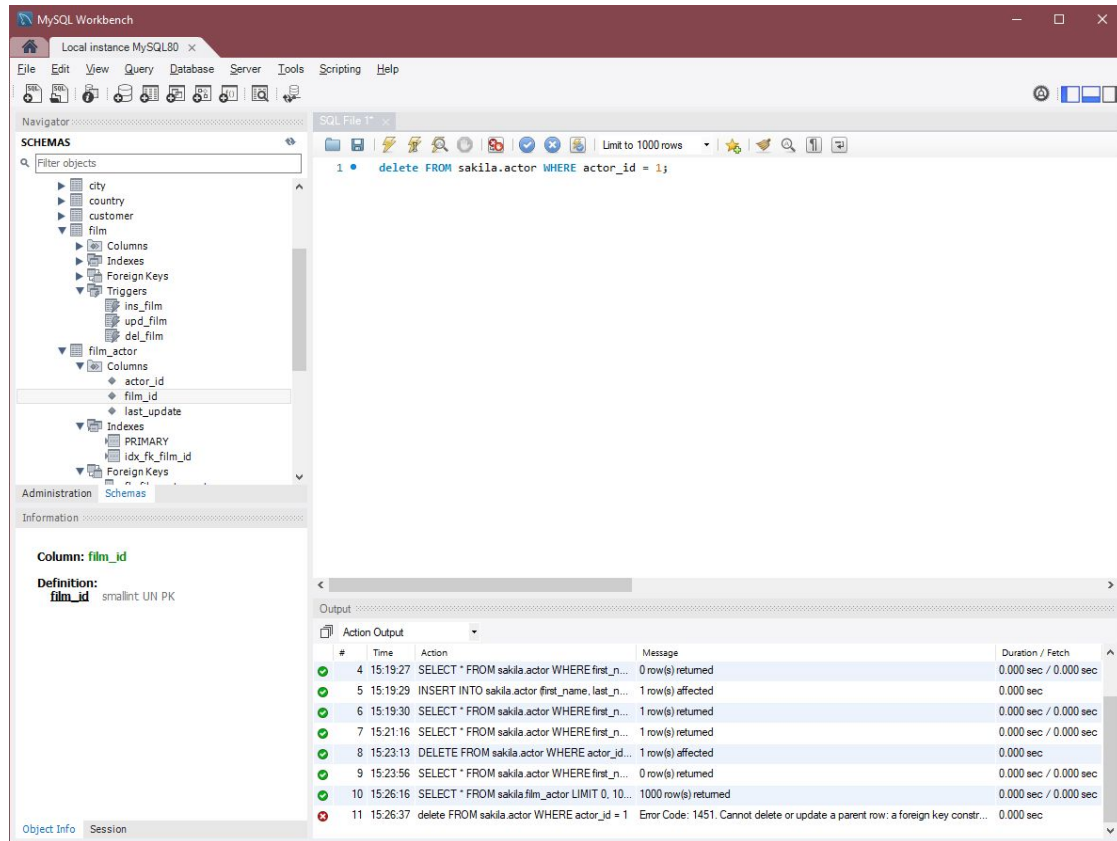
- 1- Inclua um novo dado em qualquer tabela, verifique sua existência e depois delete-o
- 2- Tente deletar um actor da tabela **actor**

# DELETE

## Dependência de dados

Ao tentar deletar um actor, podemos ver que o MySQL vai retornar um erro. Dizendo que existe uma tabela com dependência desse registro que tentamos remover.

Nesse caso, precisamos primeiro remover os registros que fazem referência e DEPOIS remover o registro “pai”



The screenshot shows the MySQL Workbench interface. The left sidebar displays the database schema, including tables like `city`, `country`, `customer`, `film`, `ins_film`, `upd_film`, `del_film`, `film_actor`, and `film_info`. The main window shows a SQL query: `delete FROM sakila.actor WHERE actor_id = 1;`. The bottom panel displays the execution output, which includes a table of actions and their results, followed by an error message.

#	Time	Action	Message	Duration / Fetch
4	15:19:27	SELECT * FROM sakila.actor WHERE first_n...	0 row(s) returned	0.000 sec / 0.000 sec
5	15:19:29	INSERT INTO sakila.actor first_name, last_n...	1 row(s) affected	0.000 sec
6	15:19:30	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec
7	15:21:16	SELECT * FROM sakila.actor WHERE first_n...	1 row(s) returned	0.000 sec / 0.000 sec
8	15:23:13	DELETE FROM sakila.actor WHERE actor_id...	1 row(s) affected	0.000 sec
9	15:23:56	SELECT * FROM sakila.actor WHERE first_n...	0 row(s) returned	0.000 sec / 0.000 sec
10	15:26:16	SELECT * FROM sakila.film_actor LIMIT 0, 10...	1000 row(s) returned	0.000 sec / 0.000 sec
11	15:26:37	delete FROM sakila.actor WHERE actor_id = 1	Error Code: 1451. Cannot delete or update a parent row: a foreign key constr...	0.000 sec

# CREATE e ALTER TABLE

Comandos que executam operações em objetos no Banco de Dados (DDL)

Existe um grupo de instruções chamado **Data Definition Language - DDL** (Linguagem de Definição de Dados). São instruções responsáveis pela definição estrutural de objetos no banco de dados. Os objetos principais são Schemas, Tabelas, Views e Sequences.

\*Essas instruções são executadas fora de transações, portanto não utilizam **COMMIT** ou **ROLLBACK**

**CREATE** (CRIAR) - É a instrução responsável por criar um novo objeto.

**ALTER** (ALTERAR) - É a instrução responsável por alterar um objeto existente.

**DROP** (DERRUBAR) - É a instrução responsável por remover um objeto.

**TRUNCATE** (ENCURTAR) - Semelhante a um **DELETE sem WHERE**. Remove todas as linhas de uma tabela. Essa opção é mais rápida e exige menos recursos.

\*Cuidado com os comandos **DROP** e **TRUNCATE** pois não é possível fazer um **ROLLBACK**, diferentemente do **DELETE**. Os dados serão removidos permanentemente.

# CREATE TABLE

Criando uma tabela no Banco de Dados

Vamos criar uma tabela no Banco de Dados.

A sintaxe básica do comando para criar uma tabela no MySQL é:

```
CREATE TABLE [schema].[nome_da_tabela] (  
[nome_da_coluna] [tipo] [constraint],  
PRIMARY KEY [nome_da_coluna_identidade]  
)
```

Exemplo:

```
CREATE TABLE sakila.vegetais (  
id_vegetal INT NOT NULL,  
nome VARCHAR(255) NOT NULL  
calorias DOUBLE(7,3)  
PRIMARY KEY (id_vegetal)  
)
```



# CREATE TABLE

Criando uma tabela no Banco de Dados

## Constraints

Uma constraint é uma restrição aplicada à uma coluna da nossa tabela

Algumas constraints comuns são:

**NOT NULL** - O Valor não pode ser vazio

**UNIQUE** - O valor deve ser único

**PRIMARY KEY** - Chave primária, ele é automaticamente uma combinação de **UNIQUE** e **NOT NULL**

**DEFAULT** - Define um valor padrão para a coluna

**CHECK** - Garante que o dado respeite uma condição para que possa ser inserido

# CREATE TABLE

Criando uma tabela no Banco de Dados

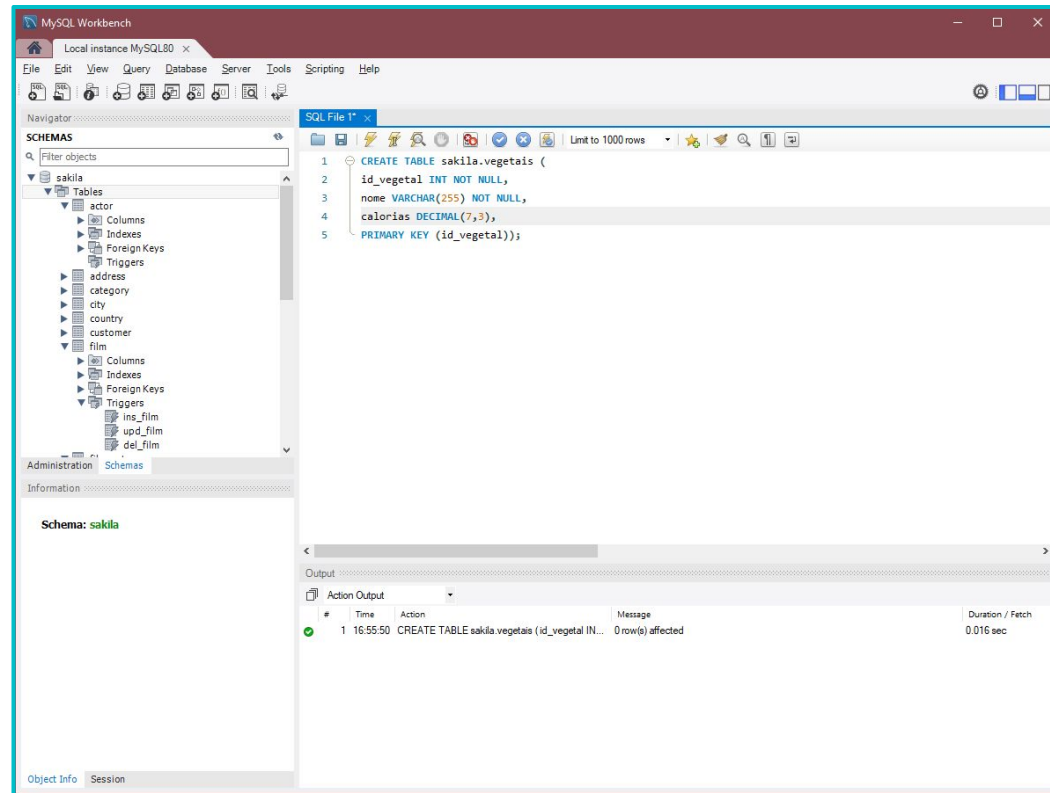
Vamos criar a nossa tabela de vegetais.

**id\_vegetal:** campo responsável por manter a identidade dos nossos registros. É um número **INTEIRO** e não pode ser null

**nome:** Campo de caracteres variáveis (String). Em SQL chamado de **VARCHAR** de tamanho 255 e não pode ser null.

**calorias:** Campo decimal contendo tamanho total 7 e precisão 3. Ou seja, ele contém 4 dígitos para parte inteira e 3 de para a decimal.

**XXXX.XXX**



# CREATE TABLE

Criando uma tabela no Banco de Dados

Podemos ver agora que temos uma tabela chamada “vegetais” no nosso banco de dados e podemos fazer um **SELECT** nela. Porém ela está vazia.

The screenshot displays the MySQL Workbench interface for a local instance of MySQL80. The left sidebar shows the 'SCHEMAS' tree with the 'sakila' database selected. The 'vegetais' table is highlighted under the 'sakila' database. The main SQL editor shows the query: `select * from sakila.vegetais;`. The output grid at the bottom shows the results of the query, indicating that 0 rows were returned.

#	Time	Action	Message	Duration / Fetch
1	17:06:10	select * from sakila.vegetais LIMIT 0, 1000	0 row(s) returned	0.000 sec / 0.000 sec

# ALTER TABLE

Alterando uma tabela no Banco de Dados

A sintaxe para um comando de alteração é:

**ALTER TABLE** [schema.tabela]  
**[ADD] | [DROP COLUMN] [MODIFY COLUMN]**

Podemos executar qualquer um dos três comandos listados quando alteramos uma tabela.

Adicionar uma coluna com **ADD** [nome\_da\_coluna] [tipo]

Adicionar uma restrição com **ADD CONSTRAINT** [nome da constraint] [tipo de constraint]

Remover uma coluna com **DROP COLUMN** [nome\_da\_coluna]

Alterar uma coluna com **MODIFY COLUMN** [nome\_da\_coluna] [novo tipo]

# ALTER TABLE

Alterando uma tabela no Banco de Dados

Explorando um pouco mais os comandos disponíveis:

Adicionando um novo campo "cor" na tabela

```
ALTER TABLE sakila.vegetais  
ADD cor VARCHAR(50);
```

Garantindo que o campo "cor" não pode ser nulo e seja verde

```
ALTER TABLE sakila.vegetais  
MODIFY cor VARCHAR(50) NOT NULL,  
ADD CONSTRAINT deve_ser_verde CHECK (cor = 'Verde');
```

Alterando o tamanho do campo "cor" para ser um pouco maior

```
ALTER TABLE sakila.vegetais  
MODIFY cor VARCHAR(120);
```

Removendo o campo "cor" na tabela

```
ALTER TABLE sakila.vegetais  
DROP COLUMN cor;
```

# DROP TABLE

Removendo uma tabela no Banco de Dados

Não faz sentido existir uma tabela de vegetais no nosso banco de dados que gerencia um negócio de aluguel de filmes. Então vamos removê-la.

A sintaxe do comando **DROP TABLE** é:

**DROP TABLE** [schema.nome\_da\_tabela];

**DROP TABLE** sakila.vegetais;

# DDL - Data Definition Language

## Exercicio

Vamos juntos criar uma tabela chamada "Orders" para gerenciar os pedidos de novos filmes para cada loja. No nosso caso, cada filme é separado em um pedido diferente

A definição da nossa tabela é a seguinte:

order\_id **SMALLINT UNSIGNED PRIMARY KEY** (id do pedido)  
store\_id **TINYINT UNSIGNED NOT NULL** (qual loja fez o pedido)  
film\_id **SMALLINT UNSIGNED NOT NULL** (qual filme foi pedido)  
ordered\_amount **SMALLINT UNSIGNED NOT NULL** (quantos filmes foram pedidos)  
price **DECIMAL(10,3) NOT NULL** (quanto custa o pedido)  
order\_date **TIMESTAMP DEFAULT CURRENT\_TIMESTAMP**

Constraints:

fk\_store\_id **FOREIGN KEY** (store\_id) **REFERENCES** sakila.store (store\_id)  
fk\_film\_id **FOREIGN KEY** (film\_id) **REFERENCES** sakila.film (film\_id)

# DDL - Data Definition Language

Exercício - Resolvido

```
CREATE TABLE sakila.order (  
  order_id INT NOT NULL AUTO_INCREMENT,  
  store_id TINYINT(255) UNSIGNED NOT NULL,  
  film_id SMALLINT(255) UNSIGNED NOT NULL,  
  ordered_amount SMALLINT(255) UNSIGNED NOT NULL,  
  price DECIMAL(10,3) UNSIGNED NOT NULL,  
  order_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  PRIMARY KEY (order_id),  
  INDEX fk_store_id_idx (store_id ASC) VISIBLE,  
  INDEX fk_film_id_idx (film_id ASC) VISIBLE);
```

```
ALTER TABLE sakila.order  
ADD CONSTRAINT fk_store_id  
  FOREIGN KEY (store_id)  
  REFERENCES sakila.store (store_id)  
  ON DELETE RESTRICT  
  ON UPDATE CASCADE,  
ADD CONSTRAINT fk_film_id  
  FOREIGN KEY (film_id)  
  REFERENCES sakila.film (film_id)  
  ON DELETE RESTRICT  
  ON UPDATE CASCADE;
```



# DDL - Data Definition Language

## Exercício - Resolvido

Nessa primeira parte do comando, estamos criando a nossa tabela, definindo as colunas, tipos de dados e dizendo quais não podem ser vazios.

Também estamos indicando a nossa chave primária e adicionando dois índices.

Especificamente para o **MySQL** estamos indicando que as colunas store\_id e film\_id devem ser utilizadas como índices. Os índices são utilizados para otimizar as consultas. Em outros bancos de dados como Oracle e PostgreSQL isso é algo que pode não ser obrigatório.

```
CREATE TABLE sakila.order (  
  order_id INT NOT NULL AUTO_INCREMENT,  
  store_id TINYINT(255) UNSIGNED NOT NULL,  
  film_id SMALLINT(255) UNSIGNED NOT NULL,  
  ordered_amount SMALLINT(255) UNSIGNED NOT NULL,  
  price DECIMAL(10,3) UNSIGNED NOT NULL,  
  order_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  PRIMARY KEY (order_id),  
  INDEX fk_store_id_idx (store_id ASC) VISIBLE,  
  INDEX fk_film_id_idx (film_id ASC) VISIBLE);
```

# DDL - Data Definition Language

## Exercício - Resolvido

Na última seção, estamos aplicando as constraints na nossa tabela já criada através de um comando **ALTER TABLE**

Essas restrições são chamadas de “Chaves Estrangeiras - FK”

A nossa chave estrangeira existe na coluna “store\_id” e faz referência à coluna “store\_id” existente na tabela sakila.store.

Isso quer dizer que não podemos inserir uma store\_id na nossa tabela sakila.order que **NÃO EXISTA** na tabela sakila.store. O mesmo vale para um filme.

A definição **ON DELETE RESTRICT** quer dizer que ao tentarmos deletar um filme ou loja, o MySQL irá restringir essa remoção.

```
ALTER TABLE sakila.order
ADD CONSTRAINT fk_store_id
FOREIGN KEY (store_id)
REFERENCES sakila.store (store_id)
ON DELETE RESTRICT
ADD CONSTRAINT fk_film_id
FOREIGN KEY (film_id)
REFERENCES sakila.film (film_id)
ON DELETE RESTRICT
```

---

# Referências

Materiais de referência e estudo

**Documentação oficial do MySQL 8.0 (Inglês):** <https://dev.mysql.com/doc/refman/8.0/en/>

**Tutorial interativo de SQL (Inglês mas possui uma tradução automática do Google):**  
<https://www.w3schools.com/sql/>

**Desafios em vários níveis e linguagens de programação incluindo Python e SQL (Inglês):** <https://www.codewars.com>

## Encerramento Aula 2

Críticas, dúvidas, sugestões?



Evolution is our core

