SQL → Structured Query Language.
DBMS → data base managment system (MySQL)

Schema é uma estrutura lógica que organiza e define as tabelas, visões, índices, funções, procedimentos armazenados e outros objetos de banco de dados. O schema ajuda a agrupar esses objetos de forma organizada e lógica, facilitando a gestão e o acesso aos dados.

A table has rows and columns.

```
create database (database name);
use (database name);
drop database (database name);
alter database (database name) read only = 1; não pode modificar mas pode acessar os dados dentro
alter database (database name) read only = 0; reverte a ação acima
select * from employees;
rename table employees to workers;
select * from users order by name;

create table employess (
	employee_id int,
	first_name varchar(50),
	last_name varchar(50),
	hourly_pay decimal(5, 2), limite de 5 dígitos, máximo 2 casas decimais depois da vírgula
	hire_date date
);

alter table employees add phone_number varchar(50);
alter table employees rename column phone_number to email;
alter table employees modify column email varchar(100);
alter table employees modify email varchar(100) after last_name;
alter table employees modify email varchar(100) first;
alter table employees drop column email;
-----------------------------------------------------------------------------------------------------------------

insert into employees values (1, “eugene”, “krabs”, 25.50, “2023-01-29”);

insert into employees
values  (1, “eugene”, “krabs”, 25.50, “2023-01-29”),
	(2, “spongebob”, “squarepants”, 8.75, “2023-04-17”),
	(3, “patrick”, “star”, 2.50, “2023-08-03”),
	(4, “sandy”, “smith”, 12.00, “2023-12-15”);

insert into employees(employee_id, first_name, last_name) values (1, “eugene”, “krabs”);

select first_name, last_name from employees;
select last_name, first_name from employees;
select * from employees where employee_id = 1;
select * from employees where hourly_pay >= 15;
select * from employees where hire_date <= "2023-01-03";
select * from employees where employee_id != 1;
select * from employees where hire_date is null; "= null" não funciona.
```

```
select * from employees where hire_date is not null;
```

```
update employees set hourly_pay = 10.25, hire_date = "2023-01-07" where employee_id = 6;
update employees set hourly_pay = null, hire_date = null where employee_id = 6;

delete from employees where employee_id = 6;
```

```
set autocommit = off;

commit;
rollback;
```

```
create table test (
	my_date date,
	my_time time,
	my_datetime datetime,
);

insert into test values (current_date(), current_time(), now());

insert into test values (current_date() + 1, null, null);
insert into test values (current_date() - 1, null, null);
```
tomorrow
yesterday

```
create table products (
	product_id int,
	product_name varchar(25) unique,
	price decima(4, 2),
);
```

ou

```
alter table products add constraint unique(product_name);

insert into products
values (100, “hamburger”, 3.99),
	(101, “fries”, 1.89),
	(102, “soda”, 1.00),
	(103, “ice cream”, 1.49);
```

```
create table products (
	product_id int,
	product_name varchar(25) unique,
	price decima(4, 2) not null,
);
```

ou

```
alter table products modify price decimal(4, 2) not null;

insert into products values (100, “hamburger”, null);
insert into products values (100, “hamburger”, 0);
```
não pode
pode

```
create table employess (
	employee_id int,
	first_name varchar(50),
	last_name varchar(50),
	hourly_pay decimal(5, 2),
	hire_date date
	constraint chk_hourly_pay check(hourly_pay >= 10.00)
);

alter table employees add constraint chk_hourly_pay check(hourly_pay >= 10.00);
alter table employees drop check chk_hourly_pay;
```

```
insert into products
values (104, “straw”, 0.0),
	(105, “napkin”, 0.0),
	(106, “fork”, 0.0),
	(107, “spoon”, 0.0);
```

* If you are in safe mode,
you can disable by going to:
Edit -> Preferences -> SQL Editor
-> uncheck Safe Updates -> (restart)

ou

```
create table products (
	product_id int,
	product_name varchar(25) unique,
	price decima(4, 2) default 0,
);
```

ou

```
alter table employees alter price set default 0;
```

and then

```
insert into products ( product_id, product_name)
values (104, “straw”),
	(105, “napkin”),
	(106, “fork”),
	(107, “spoon”);

create table transactions (
	transaction_id int,
	amount decimal(5, 2),
	transaction_date datetime default now()
);

insert into transactions ( transactions_id, amount)
values (1, 4.99),
	(2, 2.89),
	(3, 8.37);
```

---

primary key = unique + not null
there can be only one primary key per table.

```
create table transactions (
	transaction_id int primary key,
	amount decimal(5, 2),
);

ou

alter table transactions add constraint primary key(transaction_id);

insert into transactions
values (1000, 4.99),
	(1001, 2.89),
	(1000, 3.38), não funciona
	(null, 4.99); não funciona

select amount from transactions where transaction_id = 1001;
-----------------------------------------------------------------------------------------------------------------
create table transactions (
	transaction_id int primary key auto_increment,
	amount decimal(5, 2),
);

insert into transactions (amount)
values ( 4.99),
	(2.89);

alter table transactions auto_increment = 1000;
-----------------------------------------------------------------------------------------------------------------
create table costumers (
	costumer_id int primary key auto_increment,
	first_name varchar(50),
	last_name varchar(50);
);

insert into costumers (first_name, last_name)
values  ("andy", "steele"),
	("mathew", "gray"),
	("flora", "doyle");

create table transactions (
	transaction_id int primary key auto_increment,
	costumer_id int,
	amout decimal (5, 2);
	foreign key(costumer_id) references costumers (costumer_id)
);

create table users (
	username varchar(30) unique,
	pass int (255),
	foreign key (username) references professors (username) && students (username),
	foreign key (pass) references professors (password) && students (password)
);
```

```
alter table transactions drop foreign key transaction_ibfk_1
```
transaction_ibfk_1 é o nome dado ao foreign key pelo mysql

```
alter table transactions add constraint fk_costumer_id foreign key(costumer_id) references
costumers (costumer_id)

insert into transactions (amount, costumer_id)
values  (4.99, 3),
        (2.89, 2),
        (3.38, 3);

delete from costumers where costumer_id = 3;
```
não é possível deletar uma tabela pai que funciona como foreign key para uma tabela filho.

```
insert into transactions (amount, costumer_id)
values  (4.99, null);

insert into costumers (first_name, last_name)
values  ("poppy", "puff");

select *
from transactions inner join costumers
on transactions.costumer_id =  costumers.costumer_id;
```

ou

```
select transaction_id, amount,  first_name, last_name
from transactions inner join costumers ou from transactions left join costumers ou from transactions
right join costumers
on transactions.costumer_id =  costumers.costumer_id;
```

```
select count(amount) as count/"today's transactions"
from transactions;

select max(amount) as maximum
from transactions;

select min(amount) as minimum
from transactions;

select avg(amount) as average
from transactions;

select sum(amount) as sum
from transactions;

select concat(first_name, " ", last_name) as full_name
from employees;
```

```
alter table employees add column job varchar(25) after hourly_pay;

update employees set job = "manager" where employee_id = 1;
update employees set job = "cashier" where employee_id = 2;
update employees set job = "cook" where employee_id = 3;
update employees set job = "cook" where employee_id = 4;

select * from employees where hire_date < "2023-01-05" and job = "cook";

select * from employees where job = "cashier" or job = "cook";

select * from employees where job = "manager" and job = "cashier";

select * from employees where hire_date between " 2023-01-05" and " 2023-02-05";

select * from employees where job in ("cook", "cashier");
```

---

wild card character % e _
used to substitute one or more character in a string

```
select * from employees
where first_name like "s%";
```
like when used with the where clause searches any patterns occurences.

```
select * from employees
where hire_date like "2023%";

select * from employees
where last_name like "%r";

select * from employees
where hire_date like "____-01-__";
```
underscore replesents blank letters.

```
select * from employees
where job like "_a%";
```
search any jobs which the second letter is an "a".

```
select * from employees
order by last_name asc;
```
alphabetical order/ascending asc is the default, doesn't need to write

```
select * from employees
order by last_name desc;
```
reverse alphabetical order/descending

```
select * from transactions
order by amount asc, costumer_id desc;
```

---

limit clause is used to limit the number of records
useful if you're working with a lot of data
can be used to display a large data set on pages (pagination)

```
select * from costumers
limit 2;

select * from costumers
```

```
order by last_name desc limit 1;
```

supose we are on a website where you can only display 10 results per page and there are 100 results.

```
select * from costumers
limit 10;
```
first page

```
limit 10, 10;
```
second page

```
limit 20, 10;
```
third page

and so on....

---

union combines the results of the two or more select statements

```
create table incomes (
        income_namevarchar(25)
        amount decimal(7, 2)
);

insert into incomes
values ("orders", 100000.00),
        ("merchandise", 50000.00).
        ("services", 125000.00);

create table expenses (
        income_namevarchar(25)
        amount decimal(7, 2)
);

insert into expenses
values ("wages", -250000.00),
        ("tax", -50000.00).
        ("repairs", -15000.00);

select * from income;
union
select * from expenses;

select * from employees;
union
select * from costumers;
```
union não funciona quando as tabelas tem números diferentes de colunas

```
select first_name, last_name from employees;
union
select first_name, last_name from costumers;
```
agora funciona

```
select first_name, last_name from employees;
union all
```

```sql
select first_name, last_name from costumers;
```

union não admite duplicatas, se o mesmo valor estiver em duas tabelas ele só aparecerá uma vez, union all admite dulplicatas, então o valor aparecerá quantas vezes houver

---

self join
join another copy of a table to itself
used to compare rows of the same table
helps to display a hierarchy of data

```sql
alter table costumers
add referral_id int;

update costumers
set referral_id = 1
where costumer_id = 2;

select a.costumer_id, a.first_name, a.last_name,
        concat(b. first_name, " ", b. last_name) as "referred_by"
from costumers as a
inner join costumers as b
on a.referral_id = b.costumer_id;
```

Mais explicação na próxima parte →

Essa consulta SQL usa um `INNER JOIN` para combinar registros da tabela `customers` com ela mesma (isso é conhecido como auto-relacionamento). Aqui está uma explicação detalhada do que a consulta faz:

```sql
SELECT *
FROM customers AS a
INNER JOIN customers AS b
ON a.referral_id = b.customer_id;
```

Componentes da Consulta

1. **SELECT :*
    - Seleciona todas as colunas de ambas as tabelas envolvidas na consulta.
2. **FROM customers AS a:**
    - Refere-se à tabela `customers` e a renomeia como `a` para uso na consulta. Este é um alias para facilitar a referência.
3. **INNER JOIN customers AS b:**
    - Realiza um `INNER JOIN` na tabela `customers`, mas desta vez a tabela é renomeada como `b`.
4. **ON a.referral_id = b.customer_id:**
    - Define a condição de junção. Neste caso, a condição é que a coluna `referral_id` na tabela `a` (que é um alias para `customers`) deve ser igual à coluna `customer_id` na tabela `b` (também um alias para `customers`).

O que a Consulta Faz?

- **Auto-relacionamento**: A consulta faz um auto-relacionamento, juntando a tabela `customers` consigo mesma.
- **Junção Baseada em Condição**: O `INNER JOIN` é feito com base na condição de que `a.referral_id` deve ser igual a `b.customer_id`.
- **Resultado**: O resultado da consulta retornará todas as colunas da tabela `customers`, mas para cada par de registros onde o `referral_id` de um cliente (representado como `a`) corresponde ao `customer_id` de outro cliente (representado como `b`).

Exemplo Prático

Vamos considerar uma tabela `customers` com os seguintes dados:

customer_id	name	referral_id
1	Alice	3
2	Bob	1
3	Charlie	NULL
4	Dave	1

Executando a consulta acima, o resultado será:

a.customer_id	a.name	a.referral_id	b.customer_id	b.name	b.referral_id
1	Alice	3	3	Charlie	NULL
2	Bob	1	1	Alice	3
4	Dave	1	1	Alice	3

Explicação do Resultado

1. **Alice (customer_id = 1)**: Tem `referral_id` = 3, que corresponde ao `customer_id` de Charlie.
2. **Bob (customer_id = 2)**: Tem `referral_id` = 1, que corresponde ao `customer_id` de Alice.
3. **Dave (customer_id = 4)**: Tem `referral_id` = 1, que também corresponde ao `customer_id` de Alice.

Essas correspondências mostram como os clientes estão referenciando outros clientes dentro da mesma tabela.

---

views
a virtual table based on the result-set of an SQL statement.
The fields in a view are fields from one or more real tables in the database.
They're not real tables, but can be interacted with as if they were.

```
create view employee_attendance as
select first_name, last_name
from employee;

drop view employee_attendance;
```

---

index (BTree data structure)
indexes are used to find values within a specific column more quickly
MySQL normally searches sequentially through a column
the longer the column, the more expensive the operation is
update takes more time, select take less time

```
show indexes from costumers;

create index last_name_idx
on costumers(last_name);

create index last_first_name_idx
on costumers(last_name, first_name);


alter table costumers
drop index last_name_idx;

select * from costumers
where last_name = "puff" and first_name = "poppy";
```

O termo "leftmost prefix" em MySQL refere-se a um conceito crucial no contexto dos índices compostos (índices que envolvem mais de uma coluna). Quando você cria um índice composto, MySQL pode utilizar qualquer prefixo à esquerda desse índice para otimizar consultas.

Exemplificação do Conceito

Suponha que você tenha criado um índice composto em uma tabela da seguinte forma:

```
CREATE INDEX idx_composite ON tabela (coluna1, coluna2, coluna3);
```

Neste caso, o índice `idx_composite` cobre as colunas `coluna1`, `coluna2` e `coluna3`. O conceito de "leftmost prefix" significa que MySQL pode utilizar este índice para consultas que envolvam qualquer prefixo das colunas à esquerda no índice composto:

1. **Consultas que utilizam `coluna1`:**

   ```
   SELECT * FROM tabela WHERE coluna1 = 'valor';
   ```

   MySQL pode usar o índice `idx_composite` para otimizar esta consulta.

2. **Consultas que utilizam `coluna1` e `coluna2`:**

   ```
   SELECT * FROM tabela WHERE coluna1 = 'valor' AND coluna2 = 'valor';
   ```

   MySQL pode usar o índice `idx_composite` para otimizar esta consulta também.

3. **Consultas que utilizam `coluna1`, `coluna2` e `coluna3`:**

   ```
   SELECT * FROM tabela WHERE coluna1 = 'valor' AND coluna2 = 'valor' AND
   coluna3 = 'valor';
   ```

   Novamente, MySQL pode usar o índice `idx_composite`.

No entanto, **consultas que não começam com `coluna1` não podem se beneficiar deste índice**:

4. **Consultas que utilizam apenas `coluna2`:**

   ```
   SELECT * FROM tabela WHERE coluna2 = 'valor';
   ```

   MySQL não pode usar o índice `idx_composite` para esta consulta, pois não utiliza o "leftmost prefix".

5. **Consultas que utilizam `coluna2` e `coluna3`, mas não `coluna1`:**

   ```
   SELECT * FROM tabela WHERE coluna2 = 'valor' AND coluna3 = 'valor';
   ```

   MySQL também não pode usar o índice `idx_composite` para esta consulta.

Importância do Conceito

Entender o "leftmost prefix" é importante para a otimização de consultas e para a criação eficiente de índices. Ao projetar índices compostos, você deve considerar as colunas que aparecem mais frequentemente nas cláusulas `WHERE` e colocá-las na frente do índice para garantir que o índice seja útil para o maior número possível de consultas.

---

subquery
a query within another query
query(subquery)

```
select  first_name,  last_name,  hourly_pay,
        (select avg(hourly_pay) from employees) as avg_pay
from employees;
```

```
select  first_name,  last_name,  hourly_pay
from employees
where hourly_pay > (select avg(hourly_pay) from employees);
```

```
select  first_name,  last_name
from costumers
where costumer_id in
(select distinct costumer_id
from transaction
where costumer_is is not null);
```

distinct gets rid of all duplicates

imagine que, quando a query entre parenteses é resolvida, sobram os valores (1, 2 , 3). Ficariamos com:

```
select  first_name,  last_name
from costumers
where costumer_id in (1, 2, 3);
```

é assim que essa query funciona.

Ou

```
select  first_name,  last_name
from costumers
where costumer_id not in
(select distinct costumer_id
from transaction
where costumer_is is not null);
```

not in → retornará todos os clientes que não tem transações.

---

group by = aggregate all rows by a specific column
often used with aggregate functions
ex: sum(), max(), min(), avg(), count()

```
create table transactions (
	transaction_id int primary key auto_increment,
	amount decimal(5, 2),
	customer_id int,
	order_date date,
	foreign key (customer_id) references customers(customer_id)
);

insert into transactions
values  (1000, 4.99, 3, "2023-01-01"),
	(1001, 2.89, 2, "2023-01-01"),
	(1002, 3.38, 3, "2023-01-02"),
	(1003, 4.99, 1, "2023-01-02"),
	(1004, 1.00, NULL, "2023-01-03"),
	(1005, 2.49, 4, "2023-01-03"),
	(1006, 5.48, NULL, "2023-01-03");

select sum(amount), order_date
from transactions
group by order_date;

select sum(amount), costumer_id
from transactions
group by costumer_id;

select sum(amount), costumer_id
from transactions
group by costumer_id;
where count(amount) > 1;      erro
```

the where clause doesn't work with group by, instead use having, which is the same thing

```
select sum(amount), costumer_id
from transactions
group by costumer_id;
having count(amount) > 1 and costumer_id is not null;
```

---

rollup, extension of the group by clause
produces another row and shows the grand total (super-aggregate value)

```
select sum(amount), order_date
from transactions
group by order_date with rollup;

select count(transaction_id) as "# of orders", costumer_id
from transactions
group by costumer_id with rollup;

select sum(hourly_pay) as "hourly pay", employee_id
from employees
group by employee_id with rollup;
```

---

on delete set null** = when a fk is deleted, replace fk with **null
***on delete cascade** = when fk is deleted, delete row*

```
delete from costumers
where costumer_id = 4;
```
can't delete this row because it's been used as a fk in another table

```
set foreign_key_checks = 0;
set foreign_key_checks = 1;
set foreign_key_checks = 0;
delete from costumers
where costumer_id = 4;
```
desabilita essa função
reabilita essa função, SEMPRE lembre de reabilitar

```
create table transactions (
        transaction_id int primary key auto_increment,
        amount decimal(5, 2),
        customer_id int,
        order_date date,
        foreign key (customer_id) references customers(customer_id)
         on delete set null
);
```

para atualizar uma table primeiro você precisa drop the existing fk

```
alter table transactions drop foreign key fk_costumer_id;

alter table transactions
add constraint fk_costumer_id
foreign key(costumer_id) references costumers(costumer_id)
on delete set null;
```

the same rule aplies to on delete cascade

```
alter table transactions
add constraint fk_costumer_id
foreign key(costumer_id) references costumers(costumer_id)
on delete cascade;
```

agora, quando excluimos o costumer 4 da costumers table, o id fica nulo ou some toda a row na transactions table

```
delete from costumers
where costumer_id = 4;
```
---

stored procedures = is prepared SQL code that can save great if there's a query that you write often.
Reduces network traffic
increases performance
secure, admin can grant permission to use
downside: increases memory usage of every connection

ponto e virgula (delimiter) em mysql significa o fim de um statement, porém, se tiver ; no fim de costumers e no fim de end, o programa não vai entender, so we set a new delimiter temporaly.

```
delimeter $$
create procedure get_costumers()
begin
	select * from costumers;
end$$
delimiter ;

call get_costumers();

drop procedure get_costumers();

delimeter $$
create procedure find_costumer(in id int)
begin
	select *
	from costumers
	where costumer_id = id;
end$$
delimiter ;

call find_costumer(1);

delimeter $$
create procedure find_costumer(in f_name varchar(50),
                               in l_name varchar(50))
begin
	select *
	from costumers
	where first_name = if_name and last_name = l_name;
end$$
delimiter ;

call find_costumer("larry", "lobster");
```

---

trigger = when an event happens, do something
ex: insert, update, delete
checks data, handles errors, auditing tables

```
alter table employees
add column salary decimal (10, 2) after hourly_pay;

update employees set salary = hourly_pay *2080;

create trigger before_hourly_pay_update
before update on employee
for each row
set new.salary = (new.hourly_pay *2080);

show triggers;
```

```
update employees
set hourly_pay = 50
where employee_id = 1;

update employees
set hourly_pay =  hourly_pay + 1;

create trigger before_hourly_pay_insert
before insert on employee
for each row
set new.salary = (new.hourly_pay *2080);

insert into employees
values  (6, “sheldon”, “plankton”, 10, null, "janitor", “2023-01-29”, 5);

create table expenses (
        transaction_id int primary key,
        expense_name varchar (50),
        expense_total decimal(10, 2)
);

insert into expenses
values  (1, "salaries, 3, 0),
        (2, "supplies, 3, 0),
        (3, "taxes, 3, 0);

update expenses
set expense_total = (select sum(salary) from employees)
where expense_name = "salaries";

create trigger after_salary_delete
after delete on employees
for each row
update expenses
set expense_total = expense_total – old.salary
where expense_name = "salaries";

delete from employees
where employee_id = 6;

create trigger after_salary_insert
after insert on employees
for each row
update expenses
set expense_total = expense_total + new.salary
where expense_name = "salaries";

insert into employees
values  (6, “sheldon”, “plankton”, 10, null, "janitor", “2023-01-29”, 5);

create trigger after_salary_update
```

```
after update on employees
for each row
update expenses
set expense_total = expense_total + (new.salary - old.salary)
where expense_name = "salaries";

update employees
set hourly_pay = 100
where employee_id = 1;
-----------------------------------------------------------------------------------------------------------
create database music
default character set utf8;

use music;

create table artist(
        artist_id integer not null auto_increment key,
        name varchar (255)
) engine = innoDB;

create table album(
        album_id integer not null auto_increment key,
        title varchar (255),
        artist_id integer,

        index using btree(title),

        constraint foreign key (artist_id)
        references artist (artist_id)
        on delete cascade on update cascade
) engine = innoDB;

create table genre(
        genre_id integer not null auto_increment key,
        name varchar (255)
) engine = innoDB;

create table track(
        track_id integer not null auto_increment key,
        title varchar (255),
        len integer,
        rating integer,
        count integer,
        album_id integer,
        genre_id integer,

        index using btree(title),

        constraint foreign key (album_id) references album (album_id)
        on delete cascade on update cascade,
        constraint foreign key (genre_id) references genre (genre_id)
        on delete cascade on update cascade
```

```
) engine = innoDB;
-----------------------------------------------------------------------------------------------------------
create database school
default character set utf8;

use school;

create table user(
        user_id integer not null auto_increment primary key,
        email varchar (128) unique,
        name varchar (128)
) engine = innoDB;

create table course(
        course_id integer not null auto_increment primary key,
        title varchar (128) unique
) engine = innoDB;

create table member(
        user_id integer,
        course_id integer,
        role integer,

        constraint foreign key (user_id) references users (user_id)
        on delete cascade on update cascade,
        constraint foreign key (course_id) references course (course_id)
        on delete cascade on update cascade,

        primary key (user_id, course_id)
) engine = innoDB;
-----------------------------------------------------------------------------------------------------------
```

Aqui está uma explicação detalhada das instruções fornecidas:

1. `create database misc;`:
    - Esta linha cria um novo banco de dados chamado `misc` no servidor de banco de dados. O banco de dados é um contêiner onde você pode criar tabelas, armazenar dados, e gerenciar informações relacionadas à sua aplicação.
2. `grant all on misc.* to 'fred'@'localhost' identified by 'zap';`:
    - Esta linha concede todas as permissões (indicadas por `all`) ao usuário chamado `fred` para acessar e manipular todos os objetos (tabelas, vistas, etc.) no banco de dados `misc` quando ele se conectar a partir do `localhost` (o próprio servidor onde o banco de dados está rodando).
    - O usuário `fred` será autenticado com a senha `zap`.
3. `grant all on misc.* to 'fred'@'127.0.0.1' identified by 'zap';`:

- Esta linha é semelhante à anterior, mas concede as mesmas permissões ao usuário `fred` quando ele se conectar a partir do endereço IP `127.0.0.1`, que é o endereço de loopback padrão (também referindo-se ao próprio servidor).
- A senha de autenticação também é `zap`.
- **Nota:** A distinção entre `localhost` e `127.0.0.1` existe porque o MySQL trata as conexões feitas através de nomes de host e endereços IP de forma diferente. Conceder permissões para ambos garante que o usuário `fred` possa se conectar de qualquer uma das duas maneiras.

4. `use misc;`:
    - Esta linha seleciona o banco de dados `misc` como o banco de dados ativo. Após esta linha, todas as operações SQL subsequentes serão executadas no banco de dados `misc` até que outro banco de dados seja selecionado com `USE` ou a sessão seja encerrada.

Essas instruções configuram um novo banco de dados e concedem a um usuário específico (`fred`) permissões totais para trabalhar com esse banco de dados, garantindo que ele possa se conectar tanto pelo nome do host quanto pelo endereço IP local.

O comando `ENGINE=InnoDB` em SQL é usado para especificar que a tabela que está sendo criada ou modificada deve utilizar o mecanismo de armazenamento InnoDB.

Aqui está uma explicação detalhada:

1. **Mecanismo de Armazenamento (Storage Engine)**:
    - Em bancos de dados como MySQL e MariaDB, um mecanismo de armazenamento é o componente que lida com como os dados são armazenados, recuperados, atualizados e gerenciados nas tabelas. Existem diferentes mecanismos de armazenamento disponíveis, como InnoDB, MyISAM, MEMORY, entre outros.
2. **InnoDB**:
    - **InnoDB** é um dos mecanismos de armazenamento mais usados em MySQL e MariaDB. Ele é conhecido por suas funcionalidades avançadas de integridade de dados, como suporte a transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade), chaves estrangeiras (foreign keys) e bloqueio em nível de linha (row-level locking), o que melhora a concorrência e a segurança dos dados em ambientes de múltiplos usuários.
3. **Uso do `ENGINE=InnoDB`**:
    - Ao criar uma tabela, você pode especificar que deseja usar o InnoDB como o mecanismo de armazenamento com a seguinte sintaxe:

```
sql
Copiar código
CREATE TABLE my_table (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL
) ENGINE=InnoDB;
```

- Neste exemplo, a tabela `my_table` será criada usando o mecanismo de armazenamento InnoDB, aproveitando os recursos que ele oferece, como transações seguras e suporte a chaves estrangeiras.

O InnoDB é geralmente preferido em aplicações que exigem integridade referencial e operações de banco de dados seguras, especialmente em cenários de alto volume e múltiplos acessos simultâneos.

Vamos analisar o código PHP que você forneceu linha por linha, explicando a fundo cada parte.

Código Completo:

```
php
Copiar código
require_once "pdo.php";

if (isset($_POST['name']) && isset($_POST['email']) &&
isset($_POST['password'])) {
    $sql = "INSERT INTO users (name, email, password)
            VALUES (:name, :email, :password)";
    echo("<pre>\n".$sql."\n</pre>\n");
    $stmt = $pdo->prepare($sql);
    $stmt->execute(array(
        ':name' => $_POST['name'],
        ':email' => $_POST['email'],
        ':password' => $_POST['password']
    ));
}
```

Explicação Detalhada:

1. `require_once "pdo.php";`

- **`require_once`**: Esta instrução é usada para incluir um arquivo PHP, mas com uma diferença importante: o arquivo será incluído apenas uma vez, evitando problemas de múltiplas inclusões, que poderiam causar erros, especialmente com definições de funções e classes.
- **`"pdo.php"`**: Presumivelmente, este arquivo contém a configuração de conexão com o banco de dados usando PDO (PHP Data Objects). Ele deve definir a variável `$pdo`, que é usada posteriormente no código para interagir com o banco de dados.

2. `if (isset($_POST['name']) && isset($_POST['email']) && isset($_POST['password'])) {`

- **`if (isset(...))`**: A função `isset` verifica se as variáveis fornecidas foram definidas e não são nulas. Aqui, o código está verificando se os campos `name`, `email` e `password` foram enviados através do formulário HTML usando o método `POST`.
- **Objetivo**: Esta linha garante que os dados necessários para inserir um novo usuário (`name`, `email`, e `password`) foram submetidos antes de continuar com a inserção no banco de dados.

3. **`$sql = "INSERT INTO users (name, email, password) VALUES (:name, :email, :password)";`**

- **SQL Query**: Aqui, a string de SQL é criada para inserir um novo registro na tabela `users`. A consulta SQL é parametrizada, o que significa que os valores reais dos campos são passados posteriormente para evitar SQL Injection.
- **`VALUES (:name, :email, :password)`**: Estes são **placeholders nomeados**. Eles representam os valores que serão substituídos pelos dados fornecidos através do formulário. A vantagem de usar placeholders é que o PDO escapa automaticamente os valores, ajudando a proteger contra injeção de SQL.

4. **`echo("<pre>\n".$sql."\n</pre>\n");`**

- **`echo`**: Esta linha imprime a consulta SQL gerada dentro de tags `<pre>`. Isso formata o código para ser exibido de forma legível no navegador. É útil para depuração, para ver exatamente qual SQL está sendo gerado.
- **Propósito de Depuração**: É comum em desenvolvimento incluir saídas como essa para garantir que a consulta SQL esteja formatada corretamente antes de ser executada.

5. **`$stmt = $pdo->prepare($sql);`**

- **`$pdo->prepare($sql)`**: A função `prepare` é chamada no objeto `$pdo` (que representa a conexão ao banco de dados). Ela prepara a consulta SQL para execução. O método `prepare` retorna um objeto PDOStatement, que é armazenado em `$stmt`.
- **Por que `prepare`?**: Usar `prepare` com placeholders evita SQL Injection, pois os valores reais são escapados de maneira segura antes de serem inseridos na consulta.

6. **`$stmt->execute(array(':name' => $_POST['name'], ':email' => $_POST['email'], ':password' => $_POST['password']));`**

- **`$stmt->execute(...)`**: Esta função executa a consulta preparada. Ela aceita um array que mapeia os placeholders (`:name`, `:email`, `:password`) aos valores reais recebidos do formulário (`$_POST['name']`, `$_POST['email']`, `$_POST['password']`).
- **Array Associativo**: Os valores são passados como um array associativo, onde as chaves são os nomes dos placeholders e os valores são os dados do formulário.
- **Execução Segura**: Usar `execute` com placeholders nomeados e um array associativo garante que os dados sejam adequadamente escapados e que a consulta seja segura contra injeção de SQL.

Resumo

Este código exemplifica boas práticas na manipulação de dados com PHP, como o uso de PDO para interações seguras com o banco de dados. Ele inclui uma verificação para garantir que os dados necessários estão presentes antes de tentar inseri-los no banco de dados, utiliza placeholders para evitar SQL Injection e oferece uma saída de depuração para verificar a consulta SQL antes da execução.
