Introdução a Linguagem SQL: Comandos Básicos e Avançados – Parte 1

O SQL, Structured Query Language, é a linguagem utilizada pelos Banco de Dados Relacionais (BDR) modernos, sendo ela, as vezes, a única forma de você interagir com o BD em si. Todas as telas gráficas e programas que administram os BDs, nada mais é que um tradutor da funcionalidades do SQL. Eles realizam as ações que o usuário realiza na parte gráfica e converte os comandos para o SQL interagir com o BD.

O SQL é similar ao inglês

Neste ponto, você pode estar pensando que você não é um programador e aprender uma linguagem de programação não é uma das suas coisas. Felizmente, o SQL é uma linguagem simples. Tem um número limitado de comandos, que são de leitura fácil e estruturados, quase como o inglês.

Introdução a Banco de Dados

Se você não tem a mínima noção sobre o assunto de Banco de Dados, provavelmente você vai querer ler uma postagem que fiz recentemente dando uma introdução sobre os principais termos utilizados nesta área "Fundamentos da Administração de Dados: Tabelas, Entidades, Relação, Colunas, Atributos, Linhas, Registros, Tuplas, Índices, Chaves e Relacionamentos".

Caso já tenha conhecimento, pode pular para o próximo tópico da postagem.

Suponha que você tenha um BD simples para manter o inventário

que foi projetado para uma loja de conveniência. Uma das tabelas no banco de dados pode conter os preços dos itens em suas prateleiras, indexados por números de referência únicos que identificam cada item. Provavelmente, você teria uma tabela simples com nome de “Preços”. Se você precisa identificar todos os itens que tenham um preço acima de R$ 5,00, você pode recuperar a partir do seu BD esta informação utilizando o SQL fazendo uma consulta.

Realizando CONSULTAS (SELECT)

Antes de irmos direto ao comando, vamos analisar o que queremos fazer com a tabela. Imagine que você quer “selecionar todos os números de série de sua tabela de preços acima de R$ 5,00.” Se formos transformar isto em SQL, será tão simples quanto. Veja o comando:

```
SELECT NumeroSerie
FROM precos
WHERE Preco > 5;
```

Muito simples, não? Se você ler o comando, perceberá que é similar ao que queríamos antes.

Perceba que o comando será sempre similar ao nosso exemplo. Vamos a estrutura genérica do comando.

SELECT (atributos da tabela) FROM (nome da tabela) WHERE (condição);

Onde os atributos da tabela podem ser escolhidos de qualquer forma: um único atributo, vários ou todos. Veja os exemplos abaixo e descubra o que eles fazem.

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE salario > 1000;
```

```
SELECT * FROM jogadores WHERE time = "Flamengo";
```

Já vimos como selecionar os atributos que queremos, agora

vamos dar uma olhada nas condições do WHERE. É nele que “filtramos” os atributos escolhidos. O WHERE é opcional, sendo assim você omití-lo, então você terá todos os registros com os atributos escolhidos. As condições mais comuns são:

- = (igual)
- != ou <> (diferente)
- > (maior que)
- < (menor que)
- >= (maior ou igual a)
- <= (menor ou igual a)
- LIKE (similar a)
- BETWEEN (entre X e Y)
- IN (vários valores dentro da lista)
- AND (e)
- OR (ou)
- XOR (mistura do OR com
- NOT (negação)
- IS (valores iguais)

Com exceção dos dois últimos, todos possuem a seguinte sintaxe:

<expressao1> operador <expressao2>

Veja esta tabela para ajudar a entender melhor algumas condições:

<expressao1>	**Operador**	**<expressao2>**	**Resultado**
Verdadeiro	AND	Falso	Falso
Verdadeiro	AND	Verdadeiro	Verdadeiro
Falso	AND	Verdadeiro	Falso
Falso	AND	Falso	Falso
Verdadeiro	OR	Falso	Verdadeiro
Verdadeiro	OR	Verdadeiro	Verdadeiro
Falso	OR	Verdadeiro	Verdadeiro

Falso	OR	Falso	Falso
Verdadeiro	XOR	Verdadeiro	Falso
Verdadeiro	XOR	Falso	Verdadeiro
Falso	XOR	Verdadeiro	Verdadeiro
Falso	XOR	Falso	Falso

Se a qualquer das anteriores condições lhe ante-pusermos o operador NOT o resultado da operação será o contrário ao devolvido sem o operador NOT.

O último operador denominado IS se emprega para comparar duas variáveis de tipo objeto <Objeto1> IS <Objeto2>. Este operador devolve verdadeiro se os dois objetos forem iguais.

Vou utilizar um mesmo exemplo, só variando as condições. Perceba que será fácil entender o que o comando irá retornar na consulta. Veja alguns exemplos de utilização dessas condições:

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE nome = "Diego";
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE salario > 1000;
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE salario < 7000;
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE salario >= 1000;
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE salario <= 5500;
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE nome LIKE "Di*";
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE salario BETWEEN 1000 AND 5000;
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE salario > 1000 AND salario < 5000;
```

```
SELECT nome,cargo,salario FROM tbl_funcionarios WHERE (salario > 1000 AND salario < 5000) OR (nome = "Diego" AND cargo = "Analista de Sistemas");

SELECT nome,cargo,salario FROM tbl_funcionarios WHERE NOT nome = "Diego";

SELECT nome,cargo,salario FROM tbl_funcionarios WHERE nome != "Diego";

SELECT nome,cargo,salario,cidade FROM tbl_funcionarios WHERE cidade IN("Maceió", "São Paulo", "Rio de Janeiro")

SELECT Name, Category, InStock, OnOrder FROM Books WHERE Category='Fiction' XOR InStock IS NULL BY Name;
```

Falta de exemplos não é.

CRIANDO tabelas (CREATE TABLE)

Para trabalharmos com BD, precisamos das tabelas que armazenam os dados. Para isso, precisamos criá-las, seguinte a estrutura abaixo:

```
CREATE TABLE tabela (

nome_atributo tipo_dado opções,

... ... ...,

nome_atributo tipo_dado opções);
```

Os atributos das tabelas precisam ser definidos com os seus tipos, seguindo os tipos abaixo:

Tipo de Dados	Longitude	Descrição
BINARY	1 byte	Para consultas sobre tabela anexa de produtos de banco de dados que definem um tipo de dados Binário.
BIT	1 byte	Valores Sim/Não ou True/False
BYTE	1 byte	Um valor inteiro entre 0 e 255.

COUNTER	4 bytes	Um número incrementado automaticamente (de tipo Long)
CURRENCY	8 bytes	Um inteiro escalável entre 922.337.203.685.477,5808 e 922.337.203.685.477,5807.
DATETIME	8 bytes	Um valor de data ou hora entre os anos 100 e 9999.
SINGLE	4 bytes	Um valor em ponto flutuante de precisão simples com uma classificação de – 3.402823*1038 a -1.401298*10-45 para valores negativos, 1.401298*10- 45 a 3.402823*1038 para valores positivos, e 0.
DOUBLE	8 bytes	Um valor em ponto flutuante de dupla precisão com uma classificação de – 1.79769313486232*10308 a -4.94065645841247*10-324 para valores negativos, 4.94065645841247*10-324 a 1.79769313486232*10308 para valores positivos, e 0.
SHORT	2 bytes	Um inteiro curto entre -32,768 e 32,767.
LONG	4 bytes	Um inteiro longo entre -2,147,483,648 e 2,147,483,647.
LONGTEXT	1 byte por caractere	De zero a um máximo de 1.2 gigabytes.
LONGBINARY	Segundo se necessite	De zero 1 gigabyte. Utilizado para objetos OLE.
TEXT	1 byte por caractere	De zero a 255 caracteres.

A tabela de sinônimos dos tipos de dados:

Tipo de Dado	**Sinônimos**

BINARY	VARBINARY
BIT	BOOLEAN LOGICAL LOGICAL1 YESNO
BYTE	INTEGER1
COUNTER	AUTOINCREMENT
CURRENCY	MONEY
DATETIME	DATE TIME TIMESTAMP
SINGLE	FLOAT4 IEEESINGLE REAL
DOUBLE	FLOAT FLOAT8 IEEEDOUBLE NUMBER NUMERIC
SHORT	INTEGER2 SMALLINT
LONG	INT INTEGER INTEGER4
LONGBINARY	GENERAL OLEOBJECT
LONGTEXT	LONGCHAR MEMO NOTE
TEXT	ALPHANUMERIC CHAR – CHARACTER STRING – VARCHAR
VARIANT (Não Admitido)	VALUE

Vejamos alguns exemplos de comandos prontos:

```
CREATE TABLE tblprofessor(
codprofessor INTEGER CONSTRAINT primarykey PRIMARY KEY,

nome TEXT (50),
endereco TEXT (50)
telefone TEXT (15),
nascimento DATE,
sexo TEXT (1),
ativo BIT,
observacao TEXT (100));


CREATE TABLE Cliente(
Codigo INT NOT NULL AUTO_INCREMENT,
Nome VARCHAR (60) NOT NULL,
Data_Nascimento DATE,
Telefone CHAR (8),
PRIMARY KEY (Codigo));


CREATE TABLE tblNotas(

codaluno INTEGER CONSTRAINT tblalunosFK REFERENCES tblalunos,
Codcurso INTEGER CONSTRAINT tblcursosFK REFERENCES tblcursos,
Nota INTEGER,
Ano TEXT (4),
Bimestre INTEGER);


CREATE TABLE tblalunos(

codaluno INTEGER CONSTRAINT primarykey PRIMARY KEY,
nome TEXT (50),
endereco TEXT (50)
telefone TEXT (15),
nascimento DATE,
nomepai TEXT (50),
nomemae TEXT (50),
periodo TEXT (1),
serie TEXT (10),
numero TEXT (5),
observacao TEXT (100),
sexo TEXT (1),
```

```
ativo BIT);
```

INSERINDO registros (INSERT INTO)

Parar cadastrar os dados em nossas tabelas, utilizamos a estrutura abaixo:

```
INSERT INTO nome_tabela (

nome_campo1, nome_campo2, ..., nome_campoN)

VALUES (

valor_campo1, valor_campo2, ..., valor_campoN

);
```

Vejam alguns exemplos:

```
INSERT INTO Pessoas VALUES (4,'Nilsen', 'Johan', 'Bakken 2', 'Stavanger');

INSERT INTO agenda (nome, numero) VALUES ("John Doe", "555-1234");

INSERT INTO PAÍSES VALUES ('Taiwan', 'TW', 'Ásia');

INSERT INTO DEPARTAMENTOS (NUM_DEP, NOME_DEP, ADMRDEPT) VALUES ('E31', 'ARQUITETURA', 'E01');

INSERT INTO DEPARTAMENTOS (NUM_DEP, NOME_DEP, ADMRDEPT) VALUES ('B11', 'COMPRAS', 'B01'), ('E41', 'ADMINISTRAÇÃO DE BANCO DE DADOS', 'E01');
```

DELETANDO/EXCLUINDO registros (DELETE)

O comando DELETE é responsável pela exclusão de registros que não queremos mais dentro de nossas tabelas. Segue a estrutura básica:

```
DELETE FROM nome_tabela
WHERE condição
```

Lembra que olhamos alguns operadores para ser utilizado junto ao WHERE do SELECT? Ele continua valendo aqui. Veja os exemplos:

```
DELETE FROM funcionarios WHERE id_funcionario = 86;
```

```
DELETE FROM funcionarios WHERE id_funcionario < 100;
```

```
DELETE FROM funcionarios WHERE func_nome = "Diego";
```

ATUALIZANDO os registros (UPDATE)

Quando queremos atualizar/modificar os valores de algum registro na tabela, precisamos dar um UPDATE. O WHERE continua presente! Veja a estrutura básica:

```
UPDATE tabela SET coluna = valor_novo WHERE condição
```

Alguns exemplos:

```
UPDATE funcionarios SET salario = 5000 WHERE id_funcionario = 45;
```

```
UPDATE funcionarios SET salario = 7400, nome = "Diego Macêdo" WHERE id_funcionario = 1;
```

```
UPDATE funcionarios SET salario = 545 WHERE salario = 510;
```

```
UPDATE Persons SET Address='Nissestien 67', City='Sandnes' WHERE LastName='Tjessem' AND FirstName='Jakob';
```

Dica importante para este comando, é não esquecer do WHERE, ou então ele irá alterar todos os registros da sua tabela. Afinal, você não filtrou em quais registros ele irá afetar, não é verdade?

ORDENANDO os registros de uma CONSULTA (ORDER BY)

Já aprendemos a estrutura do SELECT, então vamos acrescentar algumas funcionalidades a este comando e melhorar ainda mais

nossas consultas com o ORDER BY, responsável por ordernar nosso resultado capturado de forma crescente ou decrescente baseado nas colunas que quisermos. O WHERE é opcional. Sintaxe básica é:

SELECT atributos FROM tabela ORDER BY atributo {ASC / DESC};

Os nomes ASC e DESC são opcionais, mas por padrão o ASC é utilizado, pois ele define de forma ascendente (crescente) a lista. O DESC faz o inverso, decrescente.

Veja os exemplos:

SELECT nome, salario, cargo FROM funcionarios ORDER BY nome;

SELECT nome, salario, cargo FROM funcionarios ORDER BY nome ASC;

SELECT nome, salario, cargo FROM funcionarios ORDER BY nome DESC;

SELECT nome, salario, cargo FROM funcionarios ORDER BY nome ASC, salario DESC;

SELECT nome, salario, cargo FROM funcionarios WHERE salario > 5000 ORDER BY cargo ASC, nome ASC;

Agora que já vimos os comandos básicos para o SQL, vamos começar a tornar isto mais ~~avançado~~ interessante. Mas não se preocupe. Se você conseguiu entender bem esta base, com certeza entenderá o restante. Lembrando que estou sempre disposto a ajudar com as dúvidas.

Comandos SQL Avançados

JUNTANDO tabelas (JOIN)

Este comando serve para juntar duas ou mais tabelas, baseadas em uma relação entre algumas colunas das tabelas envolvidas, através da utilização das chaves primárias e estrangeiras.

Para trabalhar com os próximos comandos, irei demonstrar aqui algumas tabelas que irão ser utilizadas para fazer mais sentido aos resultados. Vejamos elas:

TBL_FUNCIONARIO

id_func (PK)	**nome**	**salario**	**id_depto (FK)**
1	Diego	2500	345
2	Maria	5500	456
3	João	4000	567
4	José	8000	123
5	Marcos	4500	567
6	Cristina	1000	(NULL)

TBL_DEPTO

id_depto (PK)	**nome**	**ramal**
123	Diretoria	9999
456	Gerência	8888
345	T.I.	5555
567	Comercial	6666
987	Vendas	1111

Agora vamos continuar os estudos.

Existem diferentes tipos de JOINS, os quais são:

- **JOIN** – Retorna os registros das tabelas quando pelo menos um deles se relacionam;
- **LEFT JOIN** – Retorna todos os registros da tabela da esquerda, mesmo que não exista combinação com alguma linha da tabela da direita;
- **RIGHT JOIN** – O inverso do LEFT JOIN em relação as tabelas;
- **FULL JOIN** – Retorna todos os registros, mesmo quando não existe uma relação entre as tabelas;

Vamos estudar cada um deles agora.

INNER JOIN / JOIN

Retorna os registros quando existe pelo menos uma relação entre as tabelas. Se executarmos o comando abaixo:

```
SELECT f.nome, f.salario, d.nome AS depto
FROM tbl_funcionarios AS f
INNER JOIN tbl_depto AS d
ON f.id_depto = d.id_depto
ORDER BY d.nome ASC, f.nome ASC;
```

Dica: Quando utilizamos o “AS”, ele definirá um apelido para o atributo ou tabela, o qua facilita o entendimento de onde estamos selecionando os dados. Veja que “f.nome” é o nome da tabela tbl_funcionarios, f.salario é o atributo salário da tbl_funcionarios e o d.nome é o nome do departamento da tbl_depto. Bem útil para facilitar o entendimento.

Teremos a seguinte resposta desta consulta:

nome	salario	depto
João	4000	Comercial
Marcos	4500	Comercial
José	8000	Diretoria
Maria	5500	Gerência
Diego	2500	T.I.

LEFT JOIN

Retorna todos os registros da tabela da esquerda, mesmo que não exista combinação com alguma linha da tabela da direita. Vejamos um exemplo de comando:

```
SELECT f.nome, f.salario, d.nome AS depto
FROM tbl_funcionarios AS f
LEFT JOIN tbl_depto AS d
```

```
ON f.id_depto = d.id_depto
ORDER BY f.nome ASC;
```

Teremos como resultado:

nome	salario	depto
Cristina	1000	(NULL)
Diego	2500	T.I.
João	4000	Comercial
José	8000	Diretoria
Marcos	4500	Comercial
Maria	5500	Gerência

Com esta consulta, pegamos a funcionária Cristina que não pertence a nenhum departamento.

RIGHT JOIN

Retorna todos os registros da tabela da direita, mesmo que não exista combinação com alguma linha da tabela da esquerda. Veja o comando:

```
SELECT f.nome, f.salario, d.nome AS depto
FROM tbl_funcionarios AS f
RIGHT JOIN tbl_depto AS d ON f.id_depto = d.id_depto
ORDER BY f.nome ASC
```

Resultado seria este:

nome	salario	depto
(NULL)	(NULL)	Vendas
Diego	2500	T.I.
João	4000	Comercial
José	8000	Diretoria
Marcos	4500	Comercial
Maria	5500	Gerência

Perceba que ele capturou o departamento chamado “Vendas”, mas que não tinha nenhum funcionário se relacionando com ele.

UNIÃO de tabelas (UNION/UNION ALL)

O comando de UNION é usado para combinar os resultados de duas ou mais tabelas. Perceba que cada SELECT deve-se ter o mesmo número de colunas de cada tabela, incluindo o mesmo tipo de dados, assim como devem estar na mesma ordem. Veja a sintaxe:

```
SELECT colunas FROM tabela1
UNION
SELECT colunas FROM tabela2;
```

Por padrão, o UNION captura apenas valores distintos. Caso queira que ele capture os valores repetidos, deve-se usar o UNION ALL:

```
SELECT colunas FROM tabela1
UNION ALL
SELECT colunas FROM tabela2;
```

Vejamos um exemplo:

TBL_EMPREGADOS1

E_ID	**E_Nome**
01	Hansen, Ola
02	Svendson, Tove
03	Svendson, Stephen
04	Pettersen, Kari

TBL_EMPREGADOS2

E_ID	**E_Nome**
01	Turner, Sally
02	Kent, Clark
03	Svendson, Stephen

04	Scott, Stephen

O nosso resultado ao utilizar o UNION simples:

```
SELECT E_Nome FROM tbl_empregados1
UNION
SELECT E_Nome FROM tbl_empregados2;
```

E_Nome
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Scott, Stephen

Se utilizarmos o UNION ALL, irá aparecer os valores repetidos:

```
SELECT E_Nome FROM tbl_empregados1
UNION
SELECT E_Nome FROM tbl_empregados2;
```

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Svendson, Stephen
Scott, Stephen

Constantes (Constraints)

As constantes são utilizadas para determinar o limite do tipo de dados poderão ser utilizados em uma tabela. Elas podem ser utilizadas quando criamos as tabelas (com o comando CREATE TABLE) ou após criarmos (com o ALTER TABLE).

As constantes são:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

Veremos com mais detalhes abaixo cada uma delas.

NOT NULL

Com esta constante, podemos definir que uma coluna NÃO poderá receber valores nulos, ou seja, ficar sem valor algum preenchido. Caso ele esteja definido, não poderemos inserir um registro sem ter a coluna preenchida com algo ou atualizá-la (UPDATE) sem ter um valor definido. Veja o exemplo de como utilizá-lo na criação de uma tabela:

```
CREATE TABLE Pessoas
(
id_pessoa int NOT NULL,
sobrenome varchar(255) NOT NULL,
nome varchar(255),
endereco varchar(255),
cidade varchar(255)
);
```

UNIQUE

Sua função é identificar unicamente cada registro em uma tabela do BD. Esta constante e a PRIMARY KEY são responsáveis pela unicidade dos valores de uma ou mais colunas. Toda PRIMARY KEY automaticamente tem a constante UNIQUE definida por padrão. Perceba que você pode ter várias colunas com a constante UNIQUE definida, mas somente uma PRIMARY KEY por tabela. Veja o exemplo abaixo:

```
CREATE TABLE Pessoas
(
id_pessoa int NOT NULL,
sobrenome varchar(255) NOT NULL,
nome varchar(255),
endereco varchar(255),
cidade varchar(255)
UNIQUE (id_pessoa)
);
```

Utilizando após ter a tabela criada, podemos fazer assim:

```
ALTER TABLE Pessoas
ADD UNIQUE (id_pessoa);
```

Para remover a constante UNIQUE:

```
ALTER TABLE Pessoas
DROP INDEX id_pessoa;
```

PRIMARY KEY (PK)

Já vimos aqui que a chave primaria (Primary Key – PK) é responsável por identificar unicamente cada registro de uma tabela do BD. Toda chave primária deve ter um valor único e não pode conter valores nulos (NULL). Cada tabela deve ter UMA chave primária. Veja o exemplo de como utilizamos na hora de criar uma tabela:

```
CREATE TABLE Pessoas
```

```
(
id_pessoa int NOT NULL,
sobrenome varchar(255) NOT NULL,
nome varchar(255),
endereco varchar(255),
cidade varchar(255)
PRIMARY KEY (id_pessoa)
);
```

Definindo a chave primária após ter criado a tabela:

```
ALTER TABLE Pessoas
ADD PRIMARY KEY (id_pessoa);
```

Removendo a chave primária:

```
ALTER TABLE Pessoas
DROP PRIMARY KEY;
```

FOREIGN KEY (FK)

As chaves estrangeiras são responsáveis pelos relacionamentos entre as tabelas. Se você já entendeu os comandos de JOIN visto aqui na postagem lá na parte superior, já viu a FK sendo utilizada. A FK é responsável por fazer uma referência a PK de outra tabela. Caso queira entender um pouco mais sobre relacionamentos em BD. Vejamos o exemplo na hora de criarmos uma tabela:

```
CREATE TABLE Pessoas
(
id_pessoa int NOT NULL,
sobrenome varchar(255) NOT NULL,
nome varchar(255),
endereco varchar(255),
id_cidade int
PRIMARY KEY (id_pessoa),
FOREIGN KEY (id_cidade) REFERENCES cidades(id_cidade)

);
```

A FK criada na coluna “id_cidade” faz referência a coluna

“id_cidade” da tabela “cidades”.

Para definirmos após a tabela estar criada:

```
ALTER TABLE Pessoas
ADD FOREIGN KEY (id_cidade)
REFERENCES cidades(id_cidade);
```

Para removermos esta chave estrangeira:

```
ALTER TABLE Pessoas
DROP FOREIGN KEY id_cidade;
```

CHECK

O CHECK é utilizado para limitar a faixa de valores que pode ser colocado em uma coluna. No exemplo abaixo, estamos definindo que os valores para a coluna “id_pessoa” será maior que zero:

```
CREATE TABLE Pessoas
(
id_pessoa int NOT NULL,
sobrenome varchar(255) NOT NULL,
nome varchar(255),
endereco varchar(255),
id_cidade int
CHECK (id_pessoa>0)
);
```

Alterando uma tabela já criada e inserindo o CHECK:

```
ALTER TABLE Pessoas
ADD CONSTRAINT teste
CHECK (id_pessoa>0);
```

Para removermos esta chave estrangeira:

```
ALTER TABLE Pessoas
DROP CONSTRAINT teste;
```

DEFAULT

Utilizado para inserir um valor padrão em uma coluna, caso nenhum outro seja informado na hora do INSERT.

```
CREATE TABLE Pessoas
(
id_pessoa int NOT NULL,
sobrenome varchar(255) NOT NULL,
nome varchar(255),
endereco varchar(255),
cidade int DEFAULT "Maceió",
data_cadastro date DEFAULT GETDATE()
);
```

Após a tabela criada, podemos fazer assim:

```
ALTER TABLE Pessoas
ALTER cidade SET DEFAULT "Maceió";
```

Para removermos esta chave estrangeira:

```
ALTER TABLE Pessoas
ALTER cidade DROP DEFAULT;
```

Por enquanto vamos parar por aqui, pois já é assunto demais. Mas irei continuar com outros comandos importantes em outra postagem. Abraços e não deixem de compartilhar com seus amigos nas redes sociais e assinar meu blog para ficar por dentro das atualizações constantes que eu venho fazendo ultimamente aqui.