

Competências, Habilidades e Bases Tecnológicas da disciplina de Banco de Dados II

II.3 BANCO DE DADOS II					
Função: Manipulação de dados em bancos de dados relacionais					
Classificação: Execução					
Atribuições e Responsabilidades					
Manipular dados, utilizando sistemas gerenciadores de bancos de dados relacionais.					
Valores e Atitudes					
Incentivar a criatividade.					
Desenvolver a criticidade.					
Estimular o interesse na resolução de situações-problema.					
Competência			Habilidades		
1. Efetuar operações em bancos de dados com SQL.			1.1 Executar comandos SQL para manipulação de dados.		
			1.2 Utilizar transações para garantia de integridade em bancos de dados.		
Bases Tecnológicas					
Práticas de SQL com SGBDR					
<ul style="list-style-type: none">• Linguagem de manipulação de dados (DML);• Linguagem de consulta de dados (DQL);• Transações.					
DML					
<ul style="list-style-type: none">• Inserção;• Atualização;• Exclusão.					
DQL					
<ul style="list-style-type: none">• Projeção, seleção, renomeação;• Ordenação;• Agrupamento e funções agregadas;• Junção interna;• Junções externas à esquerda e à direita;• Produto cartesiano (<i>full/cross join</i>);• União, interseção e diferença.					
Transações					
<ul style="list-style-type: none">• Operações ACID;• COMMIT e ROLLBACK.					
Carga horária (horas-aula)					
Teórica	00	Prática Profissional	60	Total	60 Horas-aula
Teórica (2,5)	00	Prática Profissional (2,5)	50	Total (2,5)	00 Horas-aula
Possibilidade de divisão de classes em turmas, conforme o item 4.8 do Plano de Curso.					

Observações a considerar:
1-) Comunicação de alunos com alunos e professores:

- Microsoft Teams;
- Criação de grupo nas redes sociais também é interessante.

2-) Uso de celulares:

Para o bom andamento das aulas, recomendo que utilizem os celulares em *vibracall*, para não atrapalhar o andamento da aula.

3-) Material das aulas:

A disciplina trabalha com **NOTAS DE AULA** que são disponibilizadas ao final de cada aula.

4-) Prazos de trabalhos e atividades:

Toda atividade solicitada terá uma data limite de entrega, de forma alguma tal data será postergada ou seja, se não for entregue até a data limite a mesma receberá menção I, isso tanto para atividades entregues de forma impressa, digital ou no Microsoft Teams.

5-) Qualidade do material de atividades:

- Impressas ou manuscritas:
Muita atenção na qualidade do que será entregue, atividades sem grampear, faltando nome e número de componentes, rasgadas, amassadas, com rebarba de folha de caderno e etc. serão desconsiderados por mim.
- Digitais:
Ao enviarem atividades para o e-mail, ou inseridas no Microsoft Teams, **SEMPRE** deverá ter o nome da atividade que está sendo enviada, e no corpo do e-mail ou da atividade deverá ter o(s) nome(s) do(s) integrante(s), sem estar desta forma a atividade será **DESCONSIDERADA**.

6-) Menções e critérios de avaliação:

Na ETEC os senhores serão avaliados por MENÇÃO, onde temos:

- MB - Muito Bom;
- B - Bom;
- R - Regular;
- I - Insatisfatório.

Cada trimestres poderemos ter as seguintes formas de avaliação:

- Avaliação teórica;
- Avaliação prática (a partir do 2º trimestre, e as turmas do 2º módulo em diante);
- Seminários;
- Trabalhos teóricos e/ou práticos;
- Assiduidade;
- Outras que se fizerem necessário.

Caso o aluno tenha alguma menção I em algum dos trimestres será aplicada uma recuperação, que poderá ser em forma de trabalho prático ou teórico.

1. Recomendações bibliográficas:

CASTRO, Eduardo. Modelagem Lógica de Dados: construção básica e simplificada. São Paulo. Ciência Moderna, Agosto 2012.

MARTELLI, Richard; FILHO, Ozeas; CABRAL, Alex. Modelagem e banco de dados. São Paulo. Senac, 2017.

PEREIRA, Paloma. Introdução a banco de dados. São Paulo. Senac, 2020.

NACIONAL, Senac. Modelagem de dados. Rio de Janeiro. Senac, 2014.

2. Revisão SQL

2.1. O MySQLWorkBench.

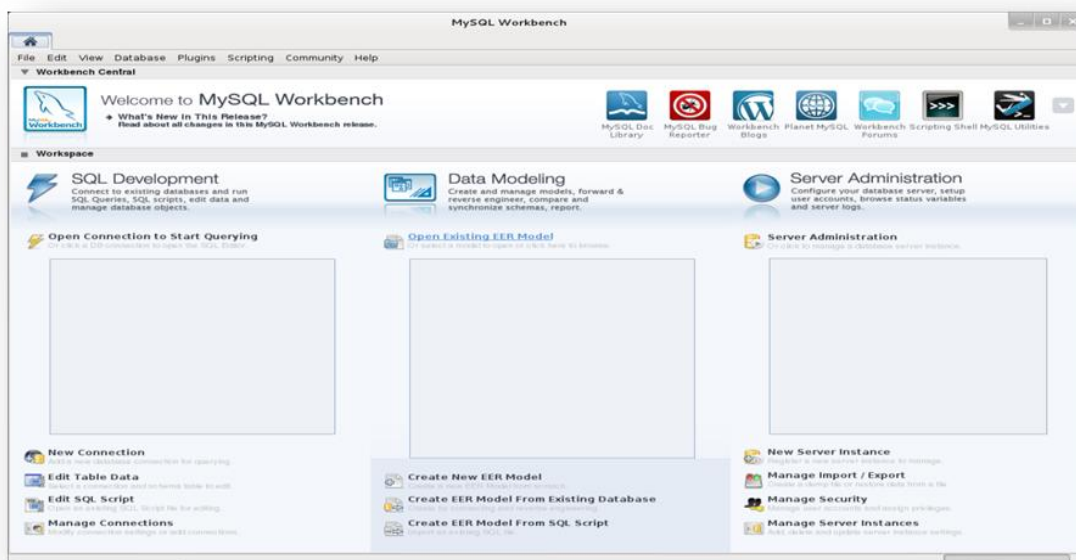
No decorrer desse curso veremos uma série de recursos de gerenciamento ligados ao Sistema Gerenciador de Banco de Dados Relacionais (**SGBDR**) "**MySQL**", esses recursos visam desde a manutenção dos bancos de dados e suas tabelas através de recursos como "**triggers**", "**stored procedures**", "**views**", "**index**" recursos esses fundamentais para a boa "**saúde**" de um banco de dados no que se refere a consistência dos dados e performance, passando também pela parte de segurança referente a criação e gestão de usuários e atribuição e revogação de direitos a esses usuários além de formas de realizar backups das bases de dados existentes.

Tais tarefas podem se mostrar desafiadoras ao “**Database Administrator**” (DBA) profissional responsável por tais tarefas, logo se faz necessário o uso de uma ou mais ferramentas que permita um alto grau de produção para esse profissional. Existe uma gama enorme de ferramentas capazes de fornecer esse auxílio para os mais diversos SGBDR corporativos existentes hoje no mercado. No nosso caso estamos trabalhando com o “**MySQL**” e vamos utilizar uma “**Integrated Development Environment**” (IDE) ou “**Ambiente Integrado de Desenvolvimento**” conhecido como “**MySQL Workbench**”.

Essa ferramenta reúne dentro de si recursos divididos em três grupos básicos são eles:

- **SQL Development** – Reúne ferramentas ligadas a codificação de comandos **SQL Data Manipulation Language** ou **Linguagem de Manipulação de Dados (DML)**, **Data definition Language** ou **Linguagem de Definição de Dados (DDL)**, **Data Control Language** ou **Linguagem de controle de Dados (DCL)**, **Data Transaction Language** ou **Linguagem de Transação de Dados (DTL)** e **Data Query Language** ou **Linguagem de Consulta de dados (DQL)**.
- **Data Modeling** – Reúne ferramentas ligadas a modelagem de banco de dados permitindo a criação de diagramas/modelos de entidade relacionamento, além disso é possível converter esses diagramas em projetos físicos (bancos de dados) através de interface gráfica e podemos também realizar processos de engenharia reversa onde podemos “apontar” para um projeto físico de um determinado banco de dados e converter esse em diagramas/modelos de entidade relacionamento isso é muito útil principalmente quando estamos trabalhando em projetos que foram legados (já existiam e nos foram passados a título de continuação) e por alguma razão qualquer a equipe anterior não documentou a base de dados ou até mesmo essa documentação foi perdida.
- **Server Administrator** – Reúne ferramentas uteis para a gerencia de backups e da “saúde” das bases de dados criadas na instancia do “**MySQL**” que se está gerenciando, permite por exemplo verificar quantas conexões estão ativas nas bases de dados existentes, verificar a “saúde” da memória principal do servidor que hospeda a instancia do “**MySQL**”, o trafego de dados, índice de eficiência das chaves, criar e associar a usuários diretos ou remover usuários e direitos, realizar backups entre outras ações.

A imagem abaixo ilustra a tela inicial do “**MySQL WorkBench**” onde podemos escolher entre os grupos de ferramentas citadas:



2.2. Aprimorando os comandos vistos

Os comandos da linguagem SQL são subdivididos em algumas categorias de comandos como: DDL, DML e DCL.

Nesta primeira etapa de BDII, usaremos a categoria DDL (*Data Definition Language* – Linguagem de Definição de Dados). Os comandos DDL são usados para definir a estrutura do banco de dados, organizando em tabelas que são compostas por campos (colunas). Comandos que compõem a DDL: CREATE, ALTER, DROP.

2.3. Tipos de Dados

Nas linguagens de programação, há quatro tipos primitivos de dados que são: inteiro, real, literal (texto) e lógico. Em cada SGBD, existem tipos de dados derivados desses tipos primitivos. No SGBD MySQL, é possível destacar os seguintes:

Tipo de Dado	Descrição
INTEGER	Representa um número inteiro
VARCHAR	Texto de tamanho variável. Máximo de 255 caracteres.
CHAR	Texto de tamanho fixo (preenche com espaços em branco os caracteres não preenchidos). Máximo de 255 caracteres.
DATE	Data
DATETIME	Data/Hora
TEXT	Texto de tamanho variável. Máximo de 65535 caracteres.
DECIMAL(p, d)	Número real, sendo que p define a precisão e d define o número de dígitos após o ponto decimal.

No MySQL, os campos ainda possuem atributos que definem a validação dos valores, tais como:

Atributo	Descrição	Aplica-se a
UNSIGNED	Sem sinal. Define que serão aceitos apenas números positivos.	Números inteiros
BINARY	Usado para diferenciar maiúsculas de minúsculas.	Texto

2.4. CREATE TABLE (com Cláusula DEFAULT)

A cláusula DEFAULT permite definir um valor padrão para um campo, que será utilizado caso não seja informado nenhum valor para esse campo na inserção de um registro na tabela.

Sintaxe:

sexo char(1) **default 'M'**

No exemplo acima, caso o campo “sexo” da tabela não seja preenchido com um valor durante a inserção de um registro, será assumido o valor ‘M’ para o campo. Para campos do tipo NUMÉRICO, o valor DEFAULT é escrito sem aspas. Exemplo:

salario decimal(10,2) **default 0**

Exemplo:

```
1 • CREATE TABLE clientes(
2     cpf integer unsigned not null,
3     nome varchar(100) not null,
4     data_nascimento date not null,
5     sexo char(1) default "M",
6     salario decimal(10,2) default 0,
7     profissao varchar(30),
8     primary key(cpf)
9 );
```

2.4.1. CONSTRAINTS

Constraints são restrições feitas para as colunas nas tabelas contendo diversos tipos, são utilizadas na criação de uma tabela ou mesmo junto com a **ALTER TABLE**, onde podemos adicionar ou remover *constraints*. Existem os seguintes:

a) **NOT NULL**: define que um campo da tabela é obrigatório (deve receber um valor na inserção de um registro);

b) **PRIMARY KEY**: define que um campo ou conjunto de campos para garantir a identidade de cada registro. Quando um campo é definido como chave primária, seu valor não pode se repetir em registros diferentes. Cada tabela só pode ter uma única chave primária. Podemos ainda as definir como:

- CHAVE PRIMÁRIA SIMPLES: composta por um único campo. Exemplo: se for definido que em um sistema de hotéis não podem existir dois clientes com o mesmo CPF, portanto este campo deverá ser definido como CHAVE PRIMÁRIA.
- CHAVE PRIMÁRIA COMPOSTA: formada por dois ou mais campos. Exemplo: se for definido em um sistema de Agências bancárias que não podem existir duas contas com o mesmo número da mesma agência, então esses dois campos formarão uma CHAVE PRIMÁRIA COMPOSTA, pois a combinação deles não pode se repetir.

Número da Conta	Número da Agência
1234	123
1234	567
3432	123

Observe que o número da conta pode se repetir individualmente, e o mesmo vale para o número da agência, porém a combinação desses dois campos não pode se repetir, garantindo que não existirão duas contas com o mesmo número na mesma agência.

Sintaxe:

- Criação de uma chave primária simples:

```
1 • CREATE TABLE contas(
2     numero integer not null primary key,
3     saldo integer default 0,
4     agencia_numero integer not null
5 );
```

Ou

```
1 • CREATE TABLE contas(
2     numero integer not null,
3     saldo integer default 0,
4     agencia_numero integer not null,
5     primary key(numero)
6 );
```

Ou

```
1 • CREATE TABLE contas(
2     numero integer not null,
3     saldo integer default 0,
4     agencia_numero integer not null,
5     constraint pk_conta primary key(numero)
6 );
```

- Criação de uma chave primária composta

```
1 • CREATE TABLE contas(
2     numero integer not null,
3     saldo integer default 0,
4     agencia_numero integer not null,
5     primary key(numero,agencia_numero)
6 );
```

Ou

```
1 • CREATE TABLE contas(
2     numero integer not null,
3     saldo integer default 0,
4     agencia_numero integer not null,
5     constraint pk_conta primary key(numero,agencia_numero)
6 );
```

c) **FOREIGN KEY**: Uma **chave estrangeira** é definida quando se deseja relacionar tabelas do banco de dados.

Sintaxe:

```
1 • CREATE TABLE contas(
2     numero integer not null,
3     saldo integer default 0,
4     agencia_numero integer not null,
5     primary key(numero,agencia_numero),
6     foreign key(agencia_numero) references agencias(numero)
7 );
```

Ou

```
1 • CREATE TABLE contas(
2     numero integer not null,
3     saldo integer default 0,
4     agencia_numero integer not null,
5     primary key(numero,agencia_numero),
6     constraint fk_contaagencia foreign key(agencia_numero) references
7     agencias(numero)
8 );
```

Na criação da chave estrangeira do exemplo anterior, pode-se ler da seguinte forma: O campo **agencia_numero** da tabela **contas** faz referência ao campo **numero** da tabela **agencias**.

d) **UNIQUE**: Uma **constraint UNIQUE** define que o valor de um campo ou de uma sequência de campos não pode se repetir em registros da mesma tabela. Essa **constraint** é criada de forma implícita quando é definida uma chave primária para uma tabela. Como só é possível ter uma chave primária por tabela, a utilização de

constraints **UNIQUE** é uma solução quando se deseja restringir valores repetidos em outros campos.

Exemplo:

```
1 • CREATE TABLE clientes(
2     cpf integer not null,
3     nome varchar(100) not null,
4     data_nascimento date not null,
5     sexo char(1) default "M",
6     salario decimal(10,2) default 0,
7     profissao varchar(30),
8     rg integer not null,
9     estado char(2) not null,
10    primary key(cpf),
11    unique(rg,estado)
12 );
```

A criação da *constraint* acima garante que um número de RG não se repetirá em um mesmo estado.

2.5.ALTER TABLE

Para não se apagar uma tabela e recriá-la, é possível fazer alterações em sua estrutura por meio do comando ALTER TABLE. Isso é importante pois a execução do comando **DROP TABLE** apaga (obviamente) todos os registros da tabela, já a execução do comando **ALTER TABLE** não exclui nenhum registro.

- Adicionar um campo

Sintaxe:

ALTER TABLE nome_da_tabela ADD nome_do_campo tipo_de_dado atributos

Exemplo:

```
1 • ALTER TABLE clientes ADD endereco varchar(90) not null;
```

- Adicionar um campo posicionando depois de um campo já existente

Sintaxe:

ALTER TABLE nome_da_tabela ADD nome_do_campo tipo_de_dado atributos AFTER nome_do_campo_ja_existente

Exemplo:

```
1 • ALTER TABLE clientes ADD endereco varchar(90) not null AFTER nome;
```

- Alterar o tipo de dado de um campo

Sintaxe:

ALTER TABLE nome_da_tabela MODIFY nome_do_campo tipo_de_dado

Exemplo:

```
1 • ALTER TABLE clientes MODIFY endereco varchar(200);
```

- Renomear um campo e modificar o tipo

Sintaxe:

ALTER TABLE nome_da_tabela CHANGE COLUMN nome_do_campo novo_nome tipo atributos;

Exemplo para mudar apenas o nome (o tipo do campo é mantido):

```
1 • ALTER TABLE clientes CHANGE COLUMN data_nascimento datanasc date;
```

Exemplo para mudar o nome e o tipo do campo:

```
1 • ALTER TABLE clientes CHANGE COLUMN data_nascimento datahoranasc datetime;
```

- Renomear uma tabela

Sintaxe:

ALTER TABLE nome_da_tabela RENAME TO novo_nome_da_tabela

Exemplo:

```
1 • ALTER TABLE clientes RENAME TO pessoas_fisicas;
```

- Apagar um campo

Sintaxe:

ALTER TABLE nome_da_tabela DROP COLUMN nome_do_campo

Exemplo:

```
1 • ALTER TABLE clientes DROP COLUMN endereco;
```

- Adicionar uma PRIMARY KEY

Sintaxe:

ALTER TABLE nome_da_tabela ADD CONSTRAINT nome_da_constraint PRIMARY KEY(campo1[,campo2,campo3,...,campoN])

Exemplo:

```
1 • ALTER TABLE clientes ADD CONSTRAINT pk_cpf PRIMARY KEY(cpf);
```

- Apagar uma PRIMARY KEY

Sintaxe:

ALTER TABLE nome_da_tabela DROP PRIMARY KEY

Ou

ALTER TABLE nome_da_tabela DROP CONSTRAINT
nome_da_constraint_da_primary_key

Exemplo:

```
1 • ALTER TABLE clientes DROP PRIMARY KEY;
```

Ou

```
1 • ALTER TABLE clientes DROP pk_cpf;
```

- Adicionar uma FOREIGN KEY

Sintaxe:

ALTER TABLE nome_da_tabela ADD CONSTRAINT nome_da_constraint FOREIGN KEY(campo1[,campo2,campo3,...,campoN]) REFERENCES
nome_da_tabela(campo1[,campo2,campo3,...,campoN]);

Exemplo:

```
1 • ALTER TABLE contas ADD CONSTRAINT fk_contaagencia  
2 FOREIGN KEY(agencia_numero) REFERENCES agencias(numero);
```

- Adicionar uma constraint UNIQUE

Sintaxe:

ALTER TABLE nome_da_tabela ADD CONSTRAINT nome_da_constraint
UNIQUE(campo1[,campo2,campo3,...,campoN])

Exemplo:

```
1 • ALTER TABLE clientes ADD CONSTRAINT un_rgestado UNIQUE(rg,estado);
```


- Apagar uma **CONSTRAINT** qualquer

Sintaxe:

ALTER TABLE nome_da_tabela DROP CONSTRAINT nome_da_constraint

Exemplo:

```
1 • ALTER TABLE contas DROP fk_contaagencia;
```

2.6. Comando *SHOW TABLES*

Para visualizar todas as tabelas em um banco de dados, utilize o comando *SHOW TABLES*.

Exemplo:

```
1 • SHOW TABLES;
```

2.7. Comando *DESC*

Para visualizar a estrutura de uma tabela, utilize o comando *DESC* (ou *DESCRIBE*).

Exemplo:

```
1 • DESC clientes;
```

3. Manipulando dados das tabelas

3.1 DML (*Data Manipulation Language* – Linguagem de Manipulação de Dados)

Os comandos DML permitem realizar operações de inserção, alteração, exclusão e seleção sobre os registros (linhas) das tabelas. Comandos que compõem a DML: *INSERT*, *UPDATE*, *DELETE* e *SELECT*. Alguns autores definem que o comando *SELECT* faz parte de uma subdivisão chamada DQL (*Data Query Language* – Linguagem de Consulta de Dados).

3.2. Comando *INSERT*

Adiciona um ou vários registros a uma tabela. Isto é referido como consulta anexação.

Sintaxe:

INSERT INTO destino [(campo1[, campo2[, ...]])]
VALUES (valor1[, valor2[, ...]])

Onde;

- **Destino** - O nome da tabela ou consulta em que os registros devem ser anexados.
- **campo1, campo2** - Os nomes dos campos aos quais os dados devem ser anexados
- **valor1, valor2** - Os valores para inserir em campos específicos do novo registro. Cada valor é inserido no campo que corresponde à posição do valor na lista: Valor1 é inserido no campo1 do novo registro, valor2 no campo2 e assim por diante.

Os valores devem ser separados com uma vírgula e os campos de textos entre aspas duplas ou simples.

Exemplo:

```
1 • INSERT INTO alunos (Id_aluno, nome, endereco, turma, turno)
2   VALUES (1, "Glaucio", "Av. das Américas", "1101", "manhã");
```

3.3. Comando *SELECT*

Permite recuperar informações existentes nas tabelas.

Sintaxe:

SELECT [DISTINCT] expressao [AS nom-atributo] [FROM from-list] [WHERE condicao]
[ORDER BY attr_name1 [ASC | DESC]

onde:

- **DISTINCT** - Para eliminar linhas duplicadas na saída.
- **Expressão** - Define os dados que queremos na saída, normalmente uma ou mais colunas de uma tabela da lista FROM.
- **AS nom-atributo** - um alias para o nome da coluna
- **FROM** - lista das tabelas na entrada

- **WHERE** - critérios da seleção
- **ORDER BY** - Critério de ordenação das tabelas de saída. Podem ser:
- **ASC** - ordem ascendente (crescente);
- **DESC** - ordem descendente (decrecente)

Exemplo:

```
1 • SELECT cidade AS cid, estado AS est FROM brasil
2 WHERE populacao > 100000 ORDER BY cidade DESC;
```

3.4 Comando *UPDATE*

Cria uma consulta atualização que altera os valores dos campos em uma tabela especificada com base em critérios específicos.

Sintaxe:

UPDATE tabela SET campo1 = valornovo, ... WHERE critério;

onde:

- **Tabela** - O nome da tabela cujos dados você quer modificar.
- **Valornovo** - Uma expressão que determina o valor a ser inserido em um campo específico nos registros atualizados.
- **critério** - Uma expressão que determina quais registros devem ser atualizados. Só os registros que satisfazem a expressão são atualizado.

Exemplo:

```
1 • UPDATE alunos SET turno = "tarde" WHERE turma = 1101;
```

UPDATE é especialmente útil quando você quer alterar muitos registros ou quando os registros que você quer alterar estão em várias tabelas. Você pode alterar vários campos ao mesmo tempo.

UPDATE não gera um conjunto de resultados. Se você quiser saber quais resultados serão alterados, examine primeiro os resultados da consulta seleção que use os mesmos critérios e então execute a consulta atualização.

3.5 Comando *DELETE*

Remove registros de uma ou mais tabelas listadas na cláusula *FROM* que satisfaz a cláusula *WHERE*.

Sintaxe:

DELETE [tabela.*] FROM tabela WHERE critério

onde:

- tabela.*** - O nome opcional da tabela da qual os registros são excluídos.
- tabela** - O nome da tabela da qual os registros são excluídos.
- critério** - Uma expressão que determina qual registro deve ser excluído.

Exemplo:

```
1 • DELETE FROM alunos WHERE turno = "Manhã";
```

DELETE é especialmente útil quando você quer excluir muitos registros. Para eliminar uma tabela inteira do banco de dados, você pode usar o método *Execute* com uma instrução *DROP*. Entretanto, se você eliminar a tabela, a estrutura é perdida. Por outro lado, quando você usa *DELETE*, apenas os dados são excluídos.

A estrutura da tabela e todas as propriedades da tabela, como atributos de campo e índices, permanecem intactos. Você pode usar *DELETE* para remover registros de tabelas que estão em uma relação um por vários com outras tabelas. Operações de exclusão em cascata fazem com que os registros das tabelas que estão no lado "vários" da relação sejam excluídos quando os registros correspondentes do lado "um" da relação são excluídos na consulta. Por exemplo, nas relações entre as tabelas Clientes e Pedidos, a tabela Clientes está do lado "um" e a tabela Pedidos está no lado "vários" da relação. Excluir um registro em Clientes faz

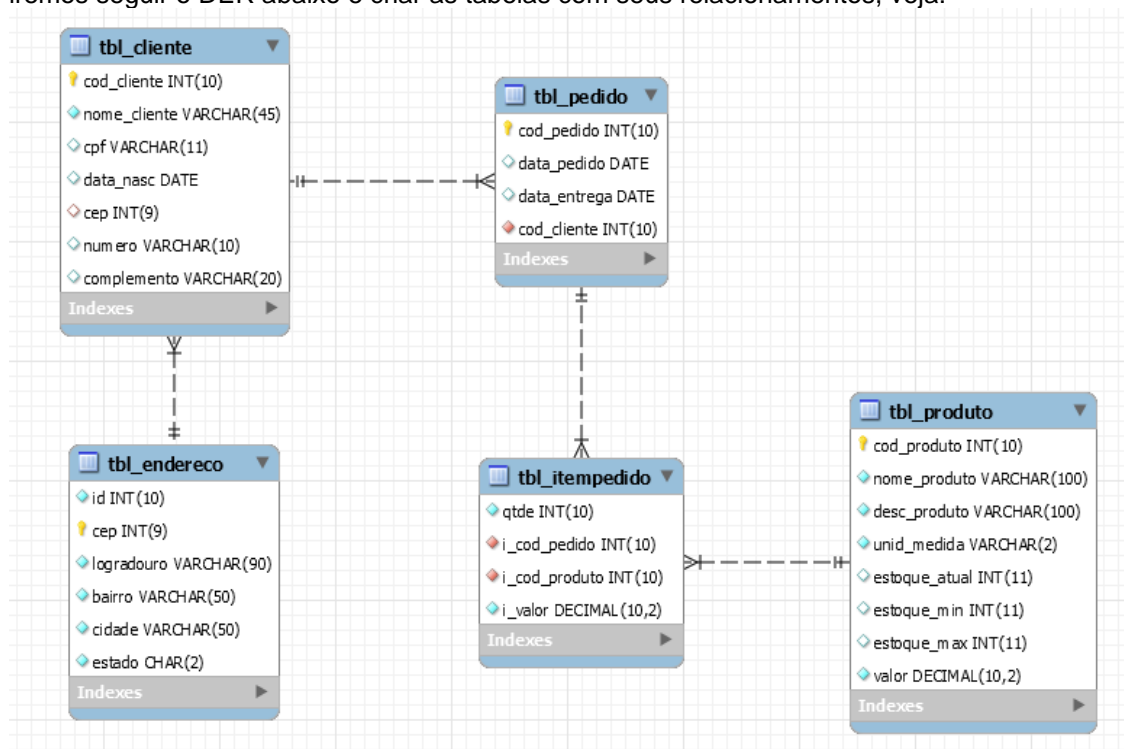
com que os registros correspondentes em Pedidos sejam excluídos se a opção de exclusão em cascata for especificada.

O **DELETE** exclui registros inteiros e não apenas dados em campos específicos. Se você quiser excluir valores de um campo específico, crie uma consulta atualização que mude os valores para *Null*.

Após remover os registros usando uma consulta exclusão, você não poderá desfazer a operação. Se quiser saber quais arquivos foram excluídos, primeiro examine os resultados de uma consulta seleção que use o mesmo critério e então, execute a consulta exclusão. Mantenha os *backups* de seus dados. Se você excluir os registros errados, poderá recuperá-los a partir dos seus backups.

4. SQL

Vamos aplicar os conceitos vistos, fazendo um banco de dados de pedidos onde iremos seguir o DER abaixo e criar as tabelas com seus relacionamentos, veja:



Mas iremos fazer esse banco de dados através de comandos SQL, primeiramente vamos organizar as coisas, iremos criar um Script SQL chamado "bd_vendas DDL" onde irá conter a criação do banco de dados e as tabelas mostradas no DER anterior.

- Criando e habilitando a base de dados:

```

/*
CRIAÇÃO DO BANCO DE DADOS BD_VENDAS - TLBD III
*/
create database bd_vendas;

-----

/*
HABILITANDO O BANCO DE DADOS PARA USO
*/
use bd_vendas;
  
```

- Criando a tabela de produtos:

```

/*
CRIAÇÃO DA TABELA DE PRODUTOS
*/
create table tbl_produto (
    cod_produto    int unsigned auto_increment,
    nome_produto   varchar(100) not null,
    desc_produto   varchar(100) not null,
    unid_medida     varchar(2) not null,
    estoque_atual  int default 0,
    estoque_min    int default 0,
    estoque_max    int default 0,
    valor          decimal(10,2) not null,
    primary key (cod_produto));

```

- Criando a tabela de endereços para armazenar os CEPs:

```

/*
CRIAÇÃO DA TABELA DE ENDEREÇO - CEP
*/
create table tbl_endereco (
    id int(10) not null,
    cep int(9) not null,
    logradouro varchar(90) not null,
    bairro varchar(50) not null,
    cidade varchar(50) not null,
    estado char(2) not null,

    constraint pk_endereco primary key (cep)
);

```

- Criando a tabela de clientes:

```

/*
CRIAÇÃO DA TABELA DE CLIENTES
*/
create table tbl_cliente (
    cod_cliente    int unsigned auto_increment,
    nome_cliente   varchar(45) not null,
    cpf            varchar(11) default '',
    data_nasc      date,
    cep            int(9) default 0,
    numero         varchar(10) default '',
    complemento    varchar(20) default '',

    primary key (cod_cliente),
    constraint foreign key fk_clientecep (cep) references tbl_endereco(cep)
);

```

- Criando a tabela de pedidos:

```

/*
CRIAÇÃO DA TABELA DE PEDIDOS
*/
create table tbl_pedido (
    cod_pedido    int unsigned auto_increment,
    data_pedido   date,
    data_entrega  date,
    cod_cliente   int unsigned not null,
    primary key (cod_pedido),
    constraint fk_cliente foreign key (cod_cliente)
    references tbl_cliente(cod_cliente));

```

- Criando a tabela de itens do pedido:

```

/*
CRIAÇÃO DA TABELA DE ITENS DO PEDIDO
*/
create table tbl_itempedido (
    qtde          int unsigned not null,
    i_cod_pedido  int unsigned not null,
    i_cod_produto int unsigned not null,
    i_valor       decimal(10,2) not null,
    constraint fk_pedido1
        foreign key (i_cod_pedido)
        references tbl_pedido (cod_pedido),
    constraint fk_tbl_produto1
        foreign key (i_cod_produto)
        references tbl_produto(cod_produto));

```

Feito isso salvem esse arquivo e mantenha sempre com vocês em nossas aulas, pois iremos incrementar mais tabelas no decorrer do curso.

Caso queiram para verificar se a construção do seu banco de dados ficou igual a mencionada inicialmente, façam a engenharia reversa (*Reverse Engineer*) no menu *DATABASE* para verificar se ficou igual.

4.1. Popular as tabelas

Iremos agora inserir dados nas tabelas para que possamos dar seguimento aos nossos estudos, mas a primeira pergunta que vem a nossa mente é: “Por qual tabela devo começar?”. Para essa resposta precisamos analisar o DER e verificar quais tabelas não possuem dependências de chaves estrangeiras e começar pelas mesmas, assim as tabelas de endereço e produto devem ser as primeiras a serem populadas, pois são tabelas que fornecem suas chaves para o preenchimento de outras, se fizermos ao contrário, a inserção dará erro.

Para isso, vamos criar outro Script SQL chamado “bd_vendas DML”, onde iremos colocar todos os *INSERTs* que iremos fazer para popular essas tabelas, esse também deverá acompanhar os senhores para nossas aulas.

Para darmos início, vamos popular a tabela de endereço com os CEPs, vou disponibilizar um Script com esses *inserts* com as cidades de Taboão da Serra e Embu das Artes, darão cerca de 2000 registros nessa tabela.

Na sequência vamos inserir os produtos:

```

insert into tbl_produto (nome_produto, desc_produto, unid_medida,
estoque_atual,estoque_min, estoque_max,valor) values
('Arroz', 'Arroz agulhinha tipo 1','SC', 10,2,20, 12.50),
('Feijão', 'Feijão carioquinha com casca','SC', 25,5,60, 7.50),
('Macarrão', 'Macarrão Adria espaguete','PC', 50,10,80, 5.50),
('Óleo', 'Óleo Lisa','LT', 15,10,45, 6.50),
('Vinagre', 'Vinagre Castelo','GR', 30,10,50, 7.89),
('Batata', 'Batata lavada','KG', 100,50,200, 4.50),
('Tomate', 'Tomate vermelho','KG', 80,8,160, 6.90),
('Cebola', 'Cebola com casca','KG', 50,5,100, 6.99),
('Leite', 'Leite Leco','CX', 25,10,90, 2.50),
('Café', 'Café do Ponto','SC', 500,100,200, 11.50);

```

Feito isso, podemos realizar as inserções dos clientes, pois sem os dados da tabela de CEPs não seria possível chegar a essa etapa. **Só lembrando que para a inserção dessa tabela, o CEP informado deverá estar cadastrado na tabela de CEPs.**

```

/*
Clientes
*/
insert into tbl_cliente(nome_cliente,cpf,data_nasc,cep,numero,complemento) values
('Marcos Costa de Sousa','12345678901','1981-02-06','6768100','1525','apto 166C'),
('Zoroastro Zoando','01987654321','1989-06-15','6757190','250',''),
('Idelbrandolância Silva','54698721364','1974-09-27','6753001','120',''),
('Cosmólio Ferreira','41368529687','1966-12-01','6753020','25','apto 255 F'),
('Conegunda Prado','54781269501','1950-10-06','6753020','50','apto 166C'),
('Brogundes Asmônio','41256398745','1940-05-10','6753400','100',''),
('Iscrência da Silva','12457965823','1974-11-25','6803040','5',''),
('Zizafânio Zizundo','54123698562','1964-08-14','6803140','25',''),
('Ricuerda Zunda','21698534589','1934-10-14','6803045','123',''),
('Aninoado Zinzão','25639856971','1976-12-25','6803070','50','');

```

4.2. Exercício para menção:

“Populem” as tabelas de acordo com o MER desenvolvido por vocês, com a massa de dados consistente de produtos e clientes, que passei para vocês anteriormente, e após isso, implementem:

- 1º - Criar no mínimo 5 pedidos;
- 2º - Para cada pedido, pelo menos 3 itens em cada;

Podem realizar em duplas, prazo para entrega, próxima semana, valendo **menção**. Irei verificar toda a codificação do banco de dados e os *INSERTs* feitos por vocês.

5. Transactions em banco de dados

Transação ou *Transaction* é uma única unidade de trabalho processada pelo Sistema de Gerenciamento de Banco de Dados (SGBD). Imagine que precisamos realizar em uma única transação duas operações de manipulação de dados (DML, do inglês Data Manipulation Language), como *INSERT*, *DELETE* e *UPDATE*. Estas operações só podem se tornar permanentes no banco de dados se todas forem executadas com sucesso. Em caso de falhas em uma das duas é possível cancelar a transação, porém todas modificações realizadas durante a mesma serão descartadas, inclusive a que obteve sucesso. Assim, os dados permanecem íntegros e consistentes.

Podemos caracterizar as transações conforme abaixo:

O BEGIN TRANSACTION indica onde ela deve começar, então os comando SQL a seguir estarão dentro desta transação.

O COMMIT TRANSACTION indica o fim normal da transação, o que tiver de comando depois já não fará parte desta transação. Neste momento tudo o que foi manipulado passa fazer parte do banco de dados normalmente e operações diversas passam enxergar o que foi feito.

O ROLLBACK TRANSACTION também fecha o bloco da transação e é a indicação que a transação deve ser terminada, mas tudo que tentou ser feito deve ser descartado porque alguma coisa errada aconteceu e ela não pode terminar normalmente. Nada realizado dentro dela será perdurado no banco de dados.

Ao contrário do que muita gente acredita *rollback* no contexto de banco de dados não significa reverter e sim voltar ao estado original. Um processo de reversão seria de complicadíssimo à impossível.

A maioria dos comandos SQL são transacionais implicitamente, ou seja, ele por si só já é uma transação. Você só precisa usar esses comandos citados quando precisa usar múltiplos comandos.

6. Comandos DML

Vamos neste momento alternar nossas aulas de acordo com o que foi visto até agora, neste capítulo iremos ver *UPDATE*, *INSERT*, *DELETE* e principalmente o *SELECT*, nos aprofundando na DQL, que é uma ramificação da DML, tudo isso para melhor fixação do conteúdo, que é vital neste momento do nosso curso. Apesar de termos as tabelas criadas e populadas conforme aula anterior, neste primeiro momento iremos nos ater em trabalhar somente com a tabela de Clientes **TBL_CLIENTE** e Endereço **TBL_ENDERECO**, em seguida passamos para as outras tabelas até finalizarmos fazendo os todos os procedimentos vistos com as chaves estrangeiras criadas nestas tabelas.

6.1 – SELECT

De acordo com a tabela “populada”, podemos retirar informações ricas da mesma, para isso iremos ver a DQL, que nos ajudará muito nesta extração de informações.

6.1.1 – CLÁUSULA DISTINCT

A cláusula DISTINCT permite que os dados repetidos de uma tabela sejam exibidos uma única vez, vamos observar o exemplo abaixo:

```
select distinct bairro
from tbl_endereco;
```

Neste caso somente serão mostrados os bairros distintos existentes na **TBL_ENDERECO**, ou seja, aparecerá no resultado 202 linhas, então podemos dizer que a tabela de endereço da nossa base de dados possui 202 bairros diferentes em seu cadastro.

Como qualquer SELECT, podemos refinar mais a consulta, por exemplo se quisermos ver a quantidade distintas de bairro por município por exemplo:

```
select distinct bairro
from tbl_endereco
where cidade = 'Taboão da Serra';
```

Nesse outro exemplo filtramos pelo município de Taboão da Serra, onde foram apresentadas 98 linhas, ou seja, em Taboão da Serra a nossa tabela possui 98 bairros diferentes.

6.1.2 – CLÁUSULA LIMIT

A cláusula LIMIT é usada para determinar um limite para a quantidade e para a faixa de linhas que podem ser retornadas, vamos neste nosso exemplo mostrar as linhas referentes aos clientes ISCRUÊNCIA e ZIZAFÂNIO observar o exemplo abaixo:

```
select *
from tbl_cliente
limit 6,2
```

Observe que o primeiro parâmetro de limite especificado (6) refere-se ao deslocamento, ou seja, ao ponto inicial, ao passo que o segundo parâmetro (2) refere-se a quantidade de linhas a serem retornadas.

Quando um único parâmetro é informado, este se refere a quantidade de linhas retornadas.

6.1.3 – CLÁUSULA ORDER BY

A cláusula ORDER BY pode ser usada juntamente com a cláusula LIMIT quando for importante que as linhas sejam retornadas de acordo com uma determinada ordem, observar o exemplo abaixo:

```
select *
from tbl_cliente
order by nome_cliente
limit 6,2;
```

Esta ordem também poderá ser ASCENDENTE ou DESCENDENTE dependendo da minha necessidade, veja:

```
select *
from tbl_cliente
order by nome_cliente desc
limit 6,2;
```

Ou

```
select *
from tbl_cliente
order by nome_cliente asc
limit 6,2;
```

Quando não informamos a classificação depois da cláusula ORDER BY por default é considerado ASC.

6.1.4 – CLÁUSULA WHERE

A cláusula *WHERE* serve para determinar quais os dados de uma tabela que serão afetados pelos comandos *SELECT*, *UPDATE* e *DELETE*, podemos dizer então que esta cláusula determina o escopo de uma consulta a algumas linhas, realizando uma filtragem dos dados que estejam de acordo com as condições definidas, vamos observar o exemplo abaixo:

```
select *
from tbl_cliente
where cpf = '12345678901';
```

Neste caso trará os dados do cliente "MARCOS COSTA DE SOUSA" que possui o CPF 12345678901.

```
select *
from tbl_endereco
where bairro = 'Jardim Caner'
```

Neste outro caso os dados de endereço que são do Bairro JD CANER serão mostrados.

```
select *
from tbl_endereco
where cidade = 'Taboão da Serra'
```

Neste outro caso os dados de endereço que são da Cidade de TABOÃO DA SERRA serão mostrados.

6.1.5 – OPERADORES AND E OR

Os operadores lógicos AND e OR são utilizados junto com a cláusula WHERE quando for necessário utilizar mais de uma condição de comparação, vamos observar o exemplo abaixo:

```
select * from tbl_endereco
where estado = 'SP'
and cidade = 'Taboão da Serra'
```

Neste caso todos endereços da UF de SP e a CIDADE de TABOÃO DA SERRA serão mostrados.

```
select * from tbl_endereco
where estado = 'SP'
or cidade = 'Taboão da Serra';
```

Neste caso todos os endereços da UF de SP ou da CIDADE de TABOÃO DA SERRA serão mostrados, em nosso exemplo irão aparecer todos os dados cadastrados, em caso de uma base de dados de todo o Brasil por exemplo, se existisse a cidade de Taboão da Serra na Bahia por exemplo, seria mostrada no resultado.

6.1.6 – OPERADOR IN

O operador IN permite checar se o valor de uma consulta está presente em uma lista de elementos.

```
select * from tbl_endereco
where bairro in ('Jardim Kuabara', 'Jardim Pazini');
```

Neste caso todos os Endereços dos BAIRROS Jardim Kuabara e Jardim Pazini serão mostrados.

Com o NOT antecedido do IN, faz a operação inversa, ou seja não trará os endereços dos BAIRROS Jardim Kuabara e Jardim Pazini.

```
select * from tbl_endereco
where bairro not in ('Jardim Kuabara', 'Jardim Pazini');
```

6.1.7 – OPERADOR LIKE

O operador LIKE é empregado quando a base para realizar pesquisa forem as colunas que estão no formato char ou varchar, vamos observar o exemplo abaixo:

```
select * from tbl_cliente
where nome_cliente like 'M%'
```

Neste caso todos os clientes que possuem a letra M no início do seu nome serão mostrados.

```
select * from tbl_cliente
where nome_cliente like '%M%';
```

Neste caso todos os clientes que possuem a letra M em qualquer parte do seu nome serão mostrados.

```
select * from tbl_cliente
where nome_cliente like '%M';
```

Neste caso todos os clientes que possuem a letra M no fim do seu nome serão mostrados.

Com o NOT antecedido do LIKE, faz a operação inversa, ou seja não trará os clientes das situações acima, veja:

```
select * from tbl_cliente
where nome_cliente not like 'M%';
```

Neste caso todos os clientes que possuem a letra M no início do seu nome não serão mostrados.

```
select * from tbl_cliente
where nome_cliente not like '%M%';
```

Neste caso todos os clientes que possuem a letra M em qualquer parte do seu nome não serão mostrados.

```
select * from tbl_cliente
where nome_cliente not like '%M';
```

Neste caso todos os clientes que possuem a letra M no fim do seu nome não serão mostrados.

```
select * from tbl_cliente
where nome_cliente like 'M_R_S%';
```

Neste caso todos os clientes que possuem na 1ª posição de seus nomes letra M, na 3ª posição a letra R e na 5ª posição a letra S e não importando o restante do nome serão mostrados, nesse caso adivinhem o nome de quem aparecerá!!!

6.1.8 – OPERADOR BETWEEN

O operador BETWEEN tem a função de permitir a consulta de uma determinada faixa de valores. Este operador permite checar se o valor de uma coluna encontra-se em um determinado intervalo, vamos observar o exemplo abaixo:

```
select * from tbl_produto
where estoque_min between 10 and 100
```

Neste caso serão mostrados todos os registros que possuem o campo ESTOQUE_MIN entre 10 e 100.

Outro exemplo, podemos usar o NOT BETWEEN

```
select * from tbl_produto
where estoque_min not between 10 and 100
```

Neste caso é feito o contrário do exemplo dado, ele trará os registros que NÃO estejam com o campo ESTOQUE_MIN entre 10 e 100.

6.1.9 – CLÁUSULA COUNT

A cláusula *COUNT* serve para contarmos quantas ocorrências existe um determinado *SELECT*, vamos observar o exemplo a seguir:

```
select count(*) from tbl_produto
where estoque_min between 10 and 100
```

Neste caso é realizada a contagem de quantas ocorrências existem de acordo com o campo ESTOQUE_MIN entre 10 e 100, ele mostrará o número 6.

6.1.10 – CLÁUSULA GROUP BY

A cláusula *GROUP BY* serve para agrupar diversos registros com base em uma ou mais colunas da tabela, vamos observar o exemplo abaixo:

```
select count(*), unid_medida
from tbl_produto
group by unid_medida
```

Neste caso serão mostrados os registros agrupados pelo campo UNID_MEDIDA, aparecerão CX = 1, GR = 1, KG = 3, LT = 1, PC = 1 e SC = 3.

```
select count(*), month(data_nasc)
from tbl_cliente
where year(data_nasc) > 1930
group by month(data_nasc)
order by 2
```

Neste caso será mostrada a quantidade de ocorrências existentes de acordo com o mês do campo DATA_NASC da tabela TBL_CLIENTE, pois estamos contando e agrupando pelo mês, com o ano do mesmo campo maior que 1930.

```
select count(*) as qtde, month(data_nasc) as mes,
year(data_nasc) as ano
from tbl_cliente
where year(data_nasc) > 1930
group by month(data_nasc), year(data_nasc)
order by 2
```

Neste caso será mostrada a quantidade de ocorrências existentes igualmente ao caso anterior, só com a diferença de colocarmos o ano e agrupando o mesmo, isso nos trará mais uma linha no resultado.

6.2. Funções agregadas

Podemos resumir as informações dos dados de uma tabela através de funções agregadas, que são aplicadas sobre as colunas de uma tabela. Devemos passar como argumento de entrada uma coluna que precisamos extrair uma simples informação da mesma, um único resultado, para que assim, não seja preciso analisar informação por informação, com uso da nossa força bruta.

Você sabe como obter um valor resultante de uma simples soma dos valores de uma coluna específica?

Sabe extrair qual é o maior ou menor valor entre muitos em um conjunto?

Sabe quantas linhas uma tabela possui?

Sabe fazer a média entre um conjunto de valores?

Iremos ver 4 diferentes funções agregadas que geram diferentes tipos de resultado, são elas:

- SUM(): computa e retorna o resultado da soma de todos os valores de uma coluna informada;
- AVG(): retorna a média dos valores de uma coluna;
- MIN(): encontra o menor valor dentre aqueles contidos na coluna de entrada;
- MAX(): encontra o maior dentre todos os valores de um conjunto;

2.2.1 Função SUM()

A função soma computa a soma dos valores de uma coluna. Essa função serve para manipularmos dados numéricos. O tipo numérico do dado pode ser *integer* e *decimal*. O resultado da função SUM() será do mesmo tipo da coluna que está sendo utilizada por esta função, vamos observar o exemplo abaixo:

```
select sum(estoque_min) as soma_estoque
from tbl_produto;
```

Neste caso será mostrada a soma do campo ESTOQUE_MIN existente na tabela.

2.2.2 Função AVG()

A função AVG() computa a média entre os valores de uma coluna de dados. Assim como a função SUM(), esta função deve ser aplicada sobre dados numéricos. Porque esta função direciona a soma dos valores, da coluna informada, para uma coluna temporária e divide o resultado da soma pela quantidade de valores que foram usadas na soma. Esse resultado final pode ser de um tipo diferente do tipo da coluna usada. Por exemplo, se você aplicar AVG em uma coluna onde só contenha inteiros, o resulta será *decimal*, isso porque ocorre uma divisão antes de finalizar o processo desta função, vamos observar o exemplo abaixo:

```
select avg(estoque_max) as media_qtde_max
from tbl_produto;
```

Neste caso será mostrada a média do campo ESTOQUE_MAX existente na tabela.

2.2.3 Função MIN()

A função MIN serve para obtermos o valor mínimo existente em uma determinada coluna, vamos observar o exemplo abaixo:

```
select min(valor) as vl_min_preco
from tbl_produto;
```

Neste caso será mostrado o mínimo valor do campo VALOR existente na tabela.

2.2.4 Função MAX()

A função MAX serve para obtermos o valor máximo existente em uma determinada coluna, vamos observar o exemplo abaixo:

```
select max(valor) as vl_max_preco
from tbl_produto;
```

Neste caso será mostrado o máximo valor do campo VALOR existente na tabela.

2.3 Operações matemáticas no SELECT

Podemos realizar “contas” no próprio SELECT para agilizar o processo de consulta, vejamos os exemplos:

```
select qtde * i_valor as valor_produto
from tbl_item_pedido
where i_cod_produto = 2
and i_cod_pedido = 1
```

Neste caso será mostrado o resultado da multiplicação dos dados das colunas VALOR e QTDE.

2.4 Prática de *Querys* SQL.

Com o objetivo de aprimorar e exercitar o conteúdo visto, vamos simular que a gestão de nossa loja queira algumas informações de nosso banco de dados, tudo isso, com o objetivo de verificar se o negócio está em bom andamento, ou, de mal a pior, para isso, é pedido para nós algumas informações como vamos ver abaixo:

1º - Queremos fazer uma panfletagem na região, e gostaria de saber quantas ruas temos nos municípios que atendemos, mas quero saber por cidade, para termos a ideia de quantas pessoas precisaremos para isso.

2º - Qual é o bairro que possui mais ruas? Queria saber para colocar mais gente trabalhando na panfletagem nestes bairros.

3º - Gostaria de saber quem são os clientes que nasceram entre as décadas de 60 e 90.

4º - Qual o mês que temos mais clientes nascidos?

5º - Qual o mês que temos mais pedidos realizados?

6º - Gostaria de saber o menor preço que vendi.

7º - Gostaria de saber o código do produto que mais vendi.

8º - Gostaria de saber a soma do meu estoque atual.

9º - Gostaria de saber a média do meu estoque atual.

10º - Gostaria de saber o valor total do meu estoque atual.

Pessoal, desenvolvam estas *querys* para que possamos validar na próxima aula.

2.4.1. Correção da prática de *Querys* SQL.

Vamos fazer uma correção da prática anterior, vale lembrar, que temos outras formas de atingirmos o mesmo resultado, esta é uma das formas:

```

01 - Correção desafio 10 querys
1 #CORREÇÃO DE EXERCÍCIOS PRÁTICOS
2 #NÚMERO 01
3 • select count(*) as qtde, cidade
4 from tbl_endereco
5 group by cidade;
6
7 #NÚMERO 02
8 • select count(*) as qtde, bairro
9 from tbl_endereco
10 group by bairro
11 order by 1 desc
12 limit 5;
13
14 #NÚMERO 03
15 • select * from tbl_cliente
16 where year(data_nasc) between 1960 and 1999;
17
18 #NÚMERO 04
19 • select count(*) as qtde, month(data_nasc) mes
20 from tbl_cliente
21 group by 2
22 order by 1 desc;

```



```

23
24 #NÚMERO 05 ←
25 • select count(*) as qtde, month(data_pedido)
26 from tbl_pedido
27 group by 2
28 order by 2 desc;
29
30 #NÚMERO 06 ←
31 • select min(i_valor) as valor_minimo
32 from tbl_itepedido;
33
34 #NÚMERO 07 ←
35 • select sum(qtde) as qtde, i_cod_produto
36 from tbl_itepedido
37 group by 2
38 order by 1 desc;
39
40 #NÚMERO 08 ←
41 • select sum(estoque_atual) as estoque_atual
42 from tbl_produto;
43
44 #NÚMERO 09 ←
45 • select avg(estoque_atual) as media_estoque
46 from tbl_produto;
47
48 #NÚMERO 10 ←
49 • select sum(estoque_atual * valor) as valor_estoque
50 from tbl_produto;

```

2.5 Joins de tabelas

Quando queremos relacionar mais de uma tabela no *SELECT*, podemos contar com alguns tipos de JOINS que veremos a seguir:

2.5.1 Inner Join

É o método de junção mais conhecido e, pois retorna os registros que são comuns às duas tabelas, precisamos fazer a amarração das tabelas com as chaves respectivas, veja os exemplos:

```

select *
from tbl_cliente as c
inner join tbl_pedido as p
on c.cod_cliente = p.cod_cliente

```

Ou

```
select *
from tbl_cliente as c, tbl_pedido as p
where c.cod_cliente = p.cod_cliente
```

Ambos darão o resultado:

cod_cliente	nome_cliente	cpf	data_nasc	cep	numero	complemento	cod_pedido	data_pedido	data_entrega	cod_cliente
1	Marcos Costa de Sousa	12345678901	1981-02-06	6768100	1525	apto 166C	1	2019-01-01	2019-01-04	1
2	Zoroastro Zoando	01987654321	1989-06-15	6757190	250		2	2019-01-05	2019-01-07	2
3	Idelbrandolância Silva	54698721364	1974-09-27	6753001	120		3	2019-01-12	2019-01-15	3
5	Coneunda Prado	54781269501	1950-10-06	6753020	50	apto 166C	4	2019-01-15	2019-01-16	5
7	Iscrência da Silva	12457965823	1974-11-25	6803040	5		5	2019-01-20	2019-01-22	7

2.6.2 Left Join

Tem como resultado todos os registros que estão na tabela A (mesmo que não estejam na tabela B) e os registros da tabela B que são comuns à tabela A, precisamos fazer a amarração das tabelas com as chaves respectivas, veja os exemplos:

```
select *
from tbl_cliente as c left join tbl_pedido as p
on c.cod_cliente = p.cod_cliente
```

O resultado será esse:

cod_cliente	nome_cliente	cpf	data_nasc	cep	numero	complemento	cod_pedido	data_pedido	data_entrega	cod_cliente
1	Marcos Costa de Sousa	12345678901	1981-02-06	6768100	1525	apto 166C	1	2019-01-01	2019-01-04	1
2	Zoroastro Zoando	01987654321	1989-06-15	6757190	250		2	2019-01-05	2019-01-07	2
3	Idelbrandolância Silva	54698721364	1974-09-27	6753001	120		3	2019-01-12	2019-01-15	3
5	Coneunda Prado	54781269501	1950-10-06	6753020	50	apto 166C	4	2019-01-15	2019-01-16	5
7	Iscrência da Silva	12457965823	1974-11-25	6803040	5		5	2019-01-20	2019-01-22	7
4	Cosmólio Ferreira	41368529687	1966-12-01	6753020	25	apto 255 F	NULL	NULL	NULL	NULL
6	Broaundes Asmônio	41256398745	1940-05-10	6753400	100		NULL	NULL	NULL	NULL
8	Zizafânio Zizundo	54123698562	1964-08-14	6803140	25		NULL	NULL	NULL	NULL
9	Ricuerda Zunda	21698534589	1934-10-14	6803045	123		NULL	NULL	NULL	NULL
10	Aninoado Zinzão	25639856971	1976-12-25	6803070	50		NULL	NULL	NULL	NULL

2.6.3 Right Join

Teremos como resultado todos os registros que estão na tabela B (mesmo que não estejam na tabela A) e os registros da tabela A que são comuns à tabela B, precisamos fazer a amarração das tabelas com as chaves respectivas, veja os exemplos:

```
select *
from tbl_pedido as p right join tbl_cliente as c
on p.cod_cliente = c.cod_cliente
```

O resultado será:

cod_pedido	data_pedido	data_entrega	cod_cliente	cod_cliente	nome_cliente	cpf	data_nasc	cep	numero	complemento
1	2019-01-01	2019-01-04	1	1	Marcos Costa de Sousa	12345678901	1981-02-06	6768100	1525	apto 166C
2	2019-01-05	2019-01-07	2	2	Zoroastro Zoando	01987654321	1989-06-15	6757190	250	
3	2019-01-12	2019-01-15	3	3	Idelbrandolância Silva	54698721364	1974-09-27	6753001	120	
4	2019-01-15	2019-01-16	5	5	Coneunda Prado	54781269501	1950-10-06	6753020	50	apto 166C
5	2019-01-20	2019-01-22	7	7	Iscrência da Silva	12457965823	1974-11-25	6803040	5	
NULL	NULL	NULL	NULL	4	Cosmólio Ferreira	41368529687	1966-12-01	6753020	25	apto 255 F
NULL	NULL	NULL	NULL	6	Broaundes Asmônio	41256398745	1940-05-10	6753400	100	
NULL	NULL	NULL	NULL	8	Zizafânio Zizundo	54123698562	1964-08-14	6803140	25	
NULL	NULL	NULL	NULL	9	Ricuerda Zunda	21698534589	1934-10-14	6803045	123	
NULL	NULL	NULL	NULL	10	Aninoado Zinzão	25639856971	1976-12-25	6803070	50	

3. VIEWS (Visualizações).

3.1 Introdução.

“Views” também conhecidas em português como visualizações nada mais são do que tabelas virtuais que são criadas a partir de seleções de dados, essas tabelas virtuais reúnem apenas os campos de uma ou mais tabelas físicas que são listados em uma instrução “SELECT” associada a “VIEW”, diferente das tabelas físicas “reais” que são armazenadas em disco e são permanentes, ou seja se o servidor de dados for reinicializado tanto as tabelas quanto o seus dados permanecerão em disco as “VIEWS” são, como já foi dito virtuais, logo seus dados permanecem armazenados na memória principal do servidor de dados, isso traz algumas vantagens a mais óbvia é o acesso extremamente rápido a esses dados, logo podemos afirmar que que as “VIEWS” são um recurso altamente recomendado em casos onde essas seleções de dados são muito requisitadas o caso mais comum de aplicação é em

relatórios, e sendo consideradas como “tabelas” tudo que vimos em DQL pode ser usado em VIEWS.

Abaixo podemos ver a sintaxe de como criar uma “view”:

CREATE VIEW <NOME_DA_VIEW> **AS** <CONSULTA>;

Para praticarmos um pouco vamos criar três “views” em nossa base de dados “VENDAS” é necessário que a massa de dados que pedi no início do semestre esteja feita. Uma vez feito isso vamos criar três “views”, observe as imagens abaixo:

A primeira “view” lista clientes e pedidos:

```
#1ª VIEW - RELAÇÃO DE CLIENTE COM PEDIDO
create view vw_cliped as
select c.cod_cliente as codigo, c.nome_cliente as nome, p.cod_pedido as pedido,
       p.data_pedido as data_requisicao
from tbl_cliente c, tbl_pedido p
where c.cod_cliente = p.cod_cliente;
```

A segunda “view” lista clientes, pedidos e itens dos pedidos:

```
#2ª VIEW - RELACIONA CLIENTES, PEDIDOS E ITEM DOS PEDIDOS
create view vw_clipedprod as
select c.cod_cliente as codigo, c.nome_cliente as nome, p.cod_pedido as pedido,
       p.data_pedido as data_requisicao,
       i.i_cod_produto as produto, i.qtde
from tbl_cliente c, tbl_pedido p, tbl_itempedido i
where c.cod_cliente = p.cod_cliente
and i.i_cod_pedido = p.cod_pedido;
```

A terceira “view” lista clientes, pedidos, itens dos pedidos e descrição dos produtos:

```
#3ª VIEW - RELACIONA CLIENTES, PEDIDOS, ITENS DOS PEDIDOS E PRODUTOS
create view vw_prodtudototal as
select c.cod_cliente as codigo, c.nome_cliente as nome, p.cod_pedido as pedido,
       p.data_pedido as data_requisicao,
       i.i_cod_produto as produto, pr.nome_produto as descricao, i.qtde,
       pr.valor, i.qtde * pr.valor as totalcomprado
from tbl_cliente c, tbl_pedido p, tbl_itempedido i, tbl_produto pr
where c.cod_cliente = p.cod_cliente
and i.i_cod_pedido = p.cod_pedido
and i.i_cod_produto = pr.cod_produto;
```

Para visualização de “VIEWS” usamos o comando SELECT convencional, e para apagar uma “VIEW” usamos o comando DROP VIEW.

3.2 VIEWS com posição consolidada

Uma VIEW consolidada nada mais é que uma consulta que agrupa informações de vários registros de uma só vez, bastante utilizada para relatórios.

A view irá mostrar os produtos dos pedidos já calculados (quantidade X valor unitário), vejamos o seguinte exemplo para explicar:

```
create view vw_consolidavenda (cod_pedido, cod_produto, valor) as
select p.cod_pedido,
       i.i_cod_produto, i.qtde * pr.valor
from tbl_pedido p, tbl_itempedido i, tbl_produto pr
where i.i_cod_pedido = p.cod_pedido
and i.i_cod_produto = pr.cod_produto;
```

Na sequência se fizermos um *SELECT* nesta view teremos:

	COD_PEDIDO	COD_PRODUTO	VALOR
	11	1	22.50
	11	2	9.60
	11	3	2.50
	12	4	17.50
	13	5	18.00

Outro exemplo seria agrupar o total dos pedidos, vejamos:

```
create view vw_vendatotal (cod_pedido, total_valor) as
select p.cod_pedido, sum(i.qtde * pr.valor)
from tbl_pedido p, tbl_itempedido i, tbl_produto pr
where i.i_cod_pedido = p.cod_pedido
and i.i_cod_produto = pr.cod_produto
group by p.cod_pedido;
```

Foram agrupados os pedidos com seus respectivos totais.

Exercícios para fixação:

De acordo com o DER já confeccionado para nossa aula, criem as seguintes views:

- Que enxerguem os dados do cliente (código e nome) e pedidos (número do pedido, data do pedido e data de entrega), onde a data do pedido seja superior a 30/01/2014;
- Que enxerguem os dados do cliente (código do cliente e nome), dados do pedido (código do pedido, data do pedido e data da entrega), os dados do item do pedido (quantidade e código do produto), onde a quantidade destes produtos seja maiores de 25;
- Que enxerguem os dados do pedido (código do pedido, código do cliente), os dados do item do pedido (quantidade, código do produto e descrição do produto);
- Que enxerguem os produtos reajustados em 11,2 %, onde deverá ser mostrado o código e a descrição do produto, o valor atual e o valor reajustado.

Esta tarefa deverá ser realizada para o dia 21/10/24, verificarei as soluções em aula, atividade em duplas.

4. Triggers (Gatilhos)

4.1 Introdução

Uma **"trigger"** também conhecida em português como **"gatilho"** é uma sub-rotina alocada dentro de uma base de dados assim como os procedimentos armazenados (**stored procedure**), porém diferentes desses que precisam ser chamados para serem executados uma **"trigger"** é executada automaticamente (**disparada**) perante uma ação sofrida em uma tabela específica dentro da base de dados onde a **"trigger"** foi criada, esse disparo pode ocorrer antes ou depois da ação em questão, logo podemos afirmar que **"triggers"** são criadas em um determinado banco de dados ficando associadas a uma tabela específica e que sua execução depende de uma ação específica na tabela em questão podendo a execução da **"trigger"** ser antes ou depois da ação que a tabela sofre.

A sintaxe de um **"trigger"** pode ser observada abaixo:

DELIMITER [define um caractere delimitador]

CREATE TRIGGER <nome> <momento> <evento> ON <tabela>

FOR EACH ROW

BEGIN

<ação a ser executada pela trigger>

END

Para tornar o entendimento mais fácil vamos criar uma “**trigger**” na base de dados “**BD_VENDAS**”, a ideia da mesma é que sempre que um registro na tabela “**TBL_CLIENTE**” for inserido será gravado na tabela chamada de “**TBL_LOG**” o usuário logado no banco de dados, assim como a data e a hora em que a inclusão ocorreu para que esses dados possam ser usados posteriormente para uma possível auditoria.

Para tanto vamos antes de qualquer coisa criar a tabela “**TBL_LOG**” que vai armazenar esses dados de auditoria a imagem abaixo traz o código de criação da tabela:

```
• create table tbl_log(
    id_log int not null auto_increment primary key,
    usuario varchar(50) not null,
    dt_log date not null,
    hora time not null
);
```

Agora vamos criar a “**trigger**”, essa por sua vez deve saber o momento em que deve ser executada sendo esses antes (**BEFORE**) ou depois (**AFTER**) de uma ação, além disso, devemos também definir em qual tabela a “**trigger**” vai atuar nesse caso, a tabela “**TBL_CLIENTE**” e, seguida devemos definir qual ação será responsável pela execução da “**trigger**” no caso uma ação de “**DELETE**”, porém vale ressaltar aqui que poderia ser qualquer outra ação que a tabela possa vir a sofrer, observe a imagem abaixo:

```
• delimiter $
create trigger trg_log before delete
on tbl_cliente
for each row
begin
    insert into tbl_log
        (usuario, dt_log, hora)
    values (user(), curdate(), curtime());
end $
```

Exibindo triggers.

Assim como os procedimentos armazenados também podemos listar as “**triggers**” de nossa base de dados, observe as imagens abaixo:

```
show triggers from bd_vendas;
```

Lista todas as “**triggers**” de uma base de dados específica, nesse caso a base de dados “**BD_VENDAS**” e novamente o usuário que está operando deve ter esse direito.

Excluindo triggers.

Assim como os procedimentos armazenados nos podemos também excluir “**triggers**” que tenham sido criadas, observe a imagem abaixo:

```
drop triggers trg_log;
```

Exercícios para fixação:

- Modifique a tabela `tbl_log` acrescentando um campo onde armazene o tipo de operação realizada, sendo: “**INSERÇÃO**”, “**ATUALIZAÇÃO**” ou “**EXCLUSÃO**” e outro campo que armazene a tabela que está sendo realizadas as ações.
- De acordo com o exercício A crie uma trigger que ao atualizar e antes de qualquer ação na tabela de Pedidos;

- c) De acordo com o exercício A crie uma trigger que ao excluir e antes de qualquer ação na tabela de Produtos;
- d) De acordo com o exercício A crie uma trigger que ao inserir e depois de qualquer ação na tabela de Clientes.

Essa atividade deverá ser desenvolvida em duplas e apresentada até a próxima aula.

5. Stored Procedures (Procedimentos armazenados).

5.1 Introdução.

Procedimentos armazenados é um conjunto de comandos SQL que podem ser armazenados dentro de uma base de dados e através desses podemos realizar praticamente qualquer tarefa dentro de um SGBDR, geralmente usamos procedimentos armazenados para tarefas como inserção, seleção, alteração e exclusão de dados.

Adotar procedimentos armazenados na manipulação de bancos de dados aumenta o desempenho de uma aplicação, pois os procedimentos ficam armazenados dentro da base de dados e são compilados no momento de sua criação e isso reduz a carga de processamento e consumo de memória principal no servidor de dados.

Além de aumentar o desempenho os procedimentos armazenados também aumentam a segurança, pois toda a lógica da base de dados fica dentro dessa, observe que isso também permite um acesso transparente ao programador na hora de fazer inserções, consultas, alterações e exclusões na base, pois para realizar tais tarefas, basta apenas chamar o procedimento armazenado responsável pela tarefa desejada, isso também permite ganho de tempo na migração de plataformas de desenvolvimento ou na criação de novas aplicações que acessam a mesma base de dados sem que haja necessidade de refazer "strings" SQL. Por último o fato das "strings" SQL estarem no lado do servidor de dados diminui muito o tráfego de dados no momento da execução dessas, pois é passado apenas o nome do procedimento ao invés de um longo comando SQL.

Dado todo o quadro descrito acima fica aqui um alerta, procedimentos armazenados não são a "bala mágica" que vai resolver todos os problemas de lógica de negócios e de carga de processamentos em uma aplicação quando criamos procedimentos armazenados em excesso transferindo toda a responsabilidade da lógica do negócio de uma aplicação para base de dados estamos na verdade criando procedimentos armazenados que possuem algoritmos complexos que exigem grande carga de processamento do servidor de dados que geralmente não é concebido para esse tipo de tarefa, logo devemos ponderar o uso de procedimentos armazenados dividindo a carga de responsabilidades do processamento que está ligado à lógica do negócio da aplicação.

Procure pensar da seguinte forma quando estiver tentando definir essas responsabilidades, procedimentos armazenados servem para tornar a manipulação da base de dados transparente para a aplicação e eventualmente essas vão carregar parte da lógica de negócios, mas não são feitas para concentrar a lógica de negócios na base de dados.

5.2 Criando procedimentos armazenados.

Sintaxe:

```
#DELIMITADOR
delimiter $
#DECLARAÇÃO DA SOTRED PROCEDURE.
create procedure nome_da_procedure()
#INICIO DO CORPO DE CODIGO DA STORED PROCEDURE
begin
    #CÓDIGO SQL QUE A PROCEDURE VAI EXECUTAR
end $ #FIM DO CORPO DE CÓDIGO DA STORED PROCEDURE
```

Podemos observar na imagem acima a sintaxe de criação de um procedimento armazenado bastante simples, tudo começa com a declaração de um delimitador na linha dois (2) através do comando "DELIMITER" seguido pelo caractere delimitador que nesse caso foi usado o "\$" poderia ter sido qualquer outro delimitador, essa técnica é interessante para mostrar para a base de dados onde o procedimento armazenado começa e onde termina.

Logo em seguida vem à declaração do procedimento armazenado propriamente dito através dos comando **"CREATE"** (criar) **"PROCEDURE"** (procedimento) e o nome que se deseja dar a ao procedimento em questão, vale lembrar aqui que o ideal é usar um nome que seja relevante com a tarefa que o procedimento realiza no fim dessa linha temos uma abertura e fechamento de parênteses por enquanto não vamos utiliza-los para nada, mas é ali que são informados os parâmetros de entrada e saída de um procedimento, veremos como explorar esse recurso mais adiante.

O **"BEGIN"** e o **"END"** marcam o inicio e o fim do corpo do procedimento que é onde vamos colocar o código SQL para executar a tarefa desejada.

Exemplo prático.

```
delimiter $
create procedure prc_lista_prod()
begin
    select * from tbl_produto
    where cod_produto in (1,3,5,7);
end $
delimiter ;
```

Na imagem acima podemos ver o código de um procedimento armazenado responsável por fazer uma seleção sem critério em uma tabela chamada `tbl_produto`, observe que esse código será executado apenas uma vez na base de dados e após isso vamos chamar o procedimento armazenado através do nome que associamos a ele no momento de sua criação.

Chamando um procedimento armazenado.

```
call prc_lista_prod();
```

Para chamar um procedimento armazenado basta utilizar o comando **"call"** seguido do nome do procedimento em questão. Assim aparecendo o resultado da procedure.

	cod_produto	nome_produto	desc_produto	unid_medida	estoque_atual	estoque_min	estoque_max	valor
	1	Arroz	Arroz acaulinha tipo 1	SC	10	2	20	12.50
	3	Macarrão	Macarrão Adria espaguete	PC	50	10	80	5.50
	5	Vinaagre	Vinaagre Castelo	GR	30	10	50	7.89
	7	Tomate	Tomate vermelho	KG	80	8	160	6.90

Excluindo um procedimento armazenado.

Uma que tenhamos criado um procedimento armazenado é possível excluir tais estruturas para tal podemos lançar mão do comando **"DROP"** seguido do nome do procedimento armazenado, observe a imagem abaixo:

```
drop procedure prc_lista_prod;
```

5.2.1 Procedimentos Armazenados com parâmetros.

Muitas vezes dentro de uma aplicação precisamos manipular dados em uma base de dados onde não sabemos exatamente quais parâmetros (**dados**) o usuário deseja utilizar, para casos como esses devemos criar procedimentos armazenados mais flexíveis que são capazes de receber valores variáveis e a partir desses valores devolver uma massa de dados para o usuário. Imagine por exemplo realizar uma instrução **"DELETE"** sem um valor para a clausula **"WHERE"**, perderíamos todos os dados da tabela.

Existem três tipos de declaração de parâmetros para procedimentos armazenados:

IN – Esse é o modo padrão, quando declaramos um parâmetro como **"IN"** seu valor é protegido, ou seja, quando o programa de chamada passa um valor e esse sofrer alteração no

decorrer da execução do procedimento o valor original da variável que representa o parâmetro permanece sem sofrer alteração.

OUT – Aqui é o contrario caso a variável que representa o valor do parâmetro sofrer uma alteração com relação ao valor passado pelo programa de chamada essa alteração se reflete no valor original do parâmetro (a variável que continha o valor original) descartando o antigo pelo novo.

INOUT – Essa variação é a fusão das duas anteriores onde um programa de chamada pode passar um valor e o procedimento armazenado pode alterar esse valor e no final será devolvido o valor alterado para o programa de chamada.

Sintaxe de declaração de procedimentos armazenados com parâmetros.

```
delimiter $
3 create procedure nome_da_procedure(mode nome_parametro tipo_parametro (tamanho_parametro))
  #INICIO DO CORPO DE CÓDIGO DA STORED PROCEDURE
begin
  #CÓDIGO SQL QUE A PROCEDURE VAI EXECUTAR
end $ #FIM DO CORPO DE CÓDIGO DA STORED PROCEDURE
delimiter;
```

Observe o código entre os parenteses:

MODE – É o tipo do comportamento que o parâmetro vai utilizar (IN, OUT ou INOUT).

NOME_PARAMETRO – É o nome que a variável que representa o parametro vai utilizar.

TIPO_PARAMETRO – É o tipo de dado que o parametro vai receber e pode ser qualquer tipo de dado valido no MySQL.

TAMANHO_PARAMETRO – É o tamanho que o tipo de dado vai suportar.

Exemplo prático de procedimento armazenado com parametro IN.

```
delimiter $
create procedure prc_prod_param_in (in nome_prod varchar(50))
begin
  select * from tbl_produto where nome_produto = nome_prod;
end $
delimiter ;
```

Depois executamos a chamada da procedure com o parâmetro

```
call prc_prod_param_in('arroz');
```

E aparece o resultado:

cod_produto	nome_produto	desc_produto	unid_medida	estoque_atual	estoque_min	estoque_max	valor
1	Arroz	Arroz aduilhina tipo 1	SC	10	2	20	12.50

Exemplo prático de procedimento armazenado com parametro OUT.

```
delimiter $
create procedure lista_produto_parametro_out (out total decimal(10,2))
begin
  select sum(valor) into total
  from tbl_produto;
end $
delimiter ;
```

Depois executamos a chamada da procedure parametrizando uma variável de saída, porém a procedure será compilada no banco e não trará resultado algum.

```
call prc_prod_param_out(@total);
```

O que a procedure faz é devolver o valor associado a variável parâmetro TOTAL.

```
select @total;
```

Trazendo o resultado:

Result Grid	
	@total
	72.28

Exemplo prático de procedimento armazenado com parametro INOUT.

```
delimiter $
create procedure prc_prod_param_inout
(in codprod int, inout nome_prod char(15),
inout valor_prod decimal(10,2))
begin
select nome_produto, valor
from tbl_produto
where cod_produto = codprod;
end $
delimiter ;
```

Depois executamos a chamada da procedure passando o parâmetro de entrada, e as variáveis de saída.

```
call prc_prod_param_inout(10,@nomeprod, @valorprod);
```

O resultado sai assim:

nome_produto	valor
Café	11.50

5.3 A estrutura de decisão IF...ELSE.

Assim como em qualquer linguagem de programação dentro da SQL temos também a estrutura de decisão "IF..ELSE" que tem como finalidade testar um determinado valor ou condição e tomar uma decisão quanto a qual direção fluxo da execução do código deve seguir.

A sintaxe da estrutura "IF..ELSE":

```
IF <(condição)> THEN
    Instruções a serem executadas caso a condição seja verdadeira.
ELSEIF <(condição)> THEN
    Instruções a serem executadas caso a condição seja verdadeira.
ELSE
    Instruções a serem executadas caso a condição seja falsa.
END IF;
```

Vamos agora fazer uma procedure para fazer a inserção de dados na TBL_PRODUTO, fazendo consistência no valor do produto, vejamos:

```

delimiter $
• create procedure prc_ins_prod (in vnomeprod char(100),
                                in vvalor decimal(10,2),
                                out msg varchar(100))
begin
    declare valor decimal(10,2);
    declare erro bool;

    set erro = true;

    if (vvalor > 0) then
        set valor = vvalor;
    else
        set erro = false;
        set msg = "valor zerado, verifique!";
    end if;

    if (erro) then
        insert into tbl_produto (nome_produto, valor)
        values (vnomeprod, vvalor);
        set msg = "incluido com sucesso!";
    end if;
end $
delimiter ;

```

Esta procedure verifica o valor do produto e dependendo da situação é dado uma situação de erro ou acerto na mesma. Abaixo executamos a procedure fazendo uma inserção:

```

• call prc_ins_prod('ovo',2.55,@msg);

```

E agora verificamos a resposta da execução da procedure:

```

• select @msg;

```

Exercícios para fixação:

- Crie um procedimento armazenado que é passado o código do produto na tabela de produtos e um percentual para calcular o acréscimo ao valor desse mesmo produto, o retorno deverá ser uma mensagem informando se a operação foi feita de forma correta.
- Crie um procedimento armazenado que grave na tabela de log (Exercício d) da atividade anterior, no campo tipo de operação, informem "INS_TRIGGER" o registro de auditoria;
- DESAFIO:** agora remova a TRIGGER que você criou conforme o exercício anterior - d), montando uma nova TRIGGER, só que agora, a mesma deverá chamar a execução do procedimento armazenado criado nessa atividade no item b).

Essa atividade deverá ser entre na aula do dia 18/10/2023.

5.4 Cursor (Cursors)

Cursor é um recurso bastante interessante em bancos de dados, pois permite que seus códigos SQL façam uma varredura de uma tabela ou consulta linha-por-linha, realizando mais de uma operação se for o caso.

Na maioria das vezes, um simples SELECT exibe na tela esta varredura, trazendo todos os registros da consulta em questão. A vantagem de usar um cursor é quando, além da exibição dos dados, queremos realizar algumas operações sobre os registros. Se o volume de operações for grande, fica muito mais fácil, limpo e prático escrever o código utilizando cursor, do que uma consulta SQL.

Por exemplo: É muito mais vantajoso criar um cursor que faça a análise de cada produto de estoque, conferindo seu histórico, calculando sua previsão de vendas para o próximo mês, capturando o melhor cliente que já o comprou, etc, do que criar um SELECT absurdamente grande que talvez não consiga ainda todas as informações de forma simples.

Abaixo vamos fazer duas PROCEDURES, uma criando uma tabela temporária, e outra efetivamente usando o Cursor, iremos fazer a chamada de uma PROCEDURE dentro da outra, mostrando esse recurso, mas antes vamos verificar o conceito de tabelas temporárias:

- **Tabelas temporárias** -> Como o próprio nome sugere, são tabelas utilizadas para armazenamento provisório de dados, não são geradas fisicamente e sim em memória, ou seja, se fecharmos o Workbench ou desligarmos nossa máquina, ela deixará de existir, sua utilização é muito válida para armazenamento de dados temporários para processamentos pontuais. A sintaxe é igual a criação de tabelas físicas, só acrescenta a palavra TEMPORARY. Veja:

```
CREATE TEMPORARY TABLE [Nome da Tabela]
(
    [Campos]
)
```

Para “dropar” a tabela temporária, o comando é:
DROP TEMPORARY TABLE [Nome da Tabela]

Antes de trabalharmos em nossa práxis com cursores, vamos ver estruturas de repetição no MySQL, abaixo temos exemplos de funcionalidades dessas estruturas, caso queiram podem implementá-las a parte, nesse momento iremos captar somente o conceito das mesmas para prosseguirmos nossa aula, vejam:

5.4.1 Estrutura de repetição (WHILE, LOOP, REPEAT)

Assim como nas linguagens de programação, em PROCEDURES também temos estruturas de repetição.

- WHILE

O comando WHILE somente executa as declarações contidas em seu corpo se a condição testada retornar o valor TRUE (verdadeiro).

A sintaxe:

```
[<rótulo>:] WHILE condição DO
    declarações
END WHILE [<rótulo>];
```

Agora para exemplificar nossa abordagem, vamos criar a tabela temporária para enfatizar nosso exemplo, ela terá o objetivo de armazenar os resultados das estruturas de repetição:

```
/* TABELA TEMPORÁRIA */
create temporary table tbl_aux (
    num_exec integer not null,
    seq integer not null,
    descricao varchar(20),
    tipo_repet varchar(10),
    primary key(num_exec, seq)
);
```

Agora vamos montar a procedure com a estrutura WHILE

```

/* ESTRUTURA DE REPETIÇÃO WHILE */
delimiter $$
create procedure acum_while (in limite int)
begin
    declare contador int default 0;
    declare soma int default 0;
    declare result varchar(20);
    declare num_exec1 integer;

    select ifnull(max(num_exec) + 1,1) num_exec into num_exec1
    from tbl_aux;

    while contador < limite do
        set contador = contador + 1;
        set soma = soma + contador;
        insert into tbl_aux(num_exec, seq, descricao, tipo_repet)
            values (num_exec1, contador, soma, 'while');
    end while;

    select * from tbl_aux;
end $$
delimiter ;

```

Após isso vamos fazer a chamada da procedure:

```
call acum_while(12);
```

O Resultado será:

Result Grid				
Filter Rows:				
Export:				
	NUM_EXEC	SEQ	DESCRICAO	TIPO_REPET
▶	1	1	1	WHILE
	1	2	3	WHILE
	1	3	6	WHILE
	1	4	10	WHILE
	1	5	15	WHILE
	1	6	21	WHILE
	1	7	28	WHILE
	1	8	36	WHILE
	1	9	45	WHILE
	1	10	55	WHILE
	1	11	66	WHILE
	1	12	78	WHILE
	2	1	1	LOOP

- LOOP

O comando LOOP sofre uma consistência dentro da rotina e a mesma finaliza através do comando LEAVE, conforme abaixo, vejamos:


```

/* ESTRUTURA DE REPETIÇÃO LOOP */
delimiter $$
• create procedure acum_loop (in limite int)
begin
    declare contador int default 0;
    declare soma int default 0;
    declare result varchar(20);
    declare num_exec1 integer;

    select ifnull(max(num_exec) + 1,1) num_exec into num_exec1
    from tbl_aux;

    loop_acum: loop

        set contador = contador + 1;
        set soma = soma + contador;

        if contador >= limite then
            leave loop_acum;
        else
            insert into tbl_aux(num_exec, seq, descricao, tipo_repet)
            values (num_exec1, contador, soma, 'loop');
        end if;


    end loop loop_acum;
    select * from tbl_aux;
end $$
delimiter ;

```

Após isso vamos fazer a chamada da procedure:

```
• call acum_loop(5);
```

O resultado será:

Result Grid  Filter Rows: <input type="text"/> Export				
	NUM_EXEC	SEQ	DESCRICAO	TIPO_REPET
	2	1	1	LOOP
	2	2	3	LOOP
	2	3	6	LOOP
	2	4	10	LOOP

- REPEAT

O comando REPEAT executa as declarações contidas em seu corpo e a condição é testada somente no final da estrutura, isso quer dizer que a mesma executará pelo menos 1 vez. Vejam:

```
/* ESTRUTURA DE REPETIÇÃO REPEAT */
delimiter $$
• create procedure acum_repeat (in limite int)
begin
    declare contador int default 0;
    declare soma int default 0;
    declare result varchar(20);
    declare num_exec1 integer;

    select ifnull(max(num_exec) + 1,1) num_exec into num_exec1
    from tbl_aux;

    repeat

        set contador = contador + 1;
        set soma = soma + contador;
        insert into tbl_aux(num_exec, seq, descricao, tipo_repet)
            values (num_exec1, contador, soma, 'repeat');

    until contador >= limite
    end repeat;
    select * from tbl_aux;

end $$
delimiter ;
```

Após isso vamos fazer a chamada da procedure:

```
• call acum_repeat(4);
```

O resultado será:

Result Grid				
Filter Rows:				
	NUM_EXEC	SEQ	DESCRICAO	TIPO_REPET
	3	1	1	REPEAT
	3	2	3	REPEAT
	3	3	6	REPEAT
	3	4	10	REPEAT

Após esses exemplos e explicações das estruturas de repetição, iremos dar continuidade com nossa práxis fazendo uma procedure de ajuste de estoque, onde faremos a chamada de outra procedure para criação de tabela temporária e assim seguiremos com novos conceitos.

```
#PROCEDURE PARA CRIAÇÃO DE TABELAS TEMPORÁRIAS
delimiter @@
create procedure cria_tabela()
begin
    create temporary table if not exists tbl_auxprod (
        aux_prod integer not null,
        aux_desc varchar(20),
        qtde_ajust varchar(10),
        data_hora datetime default now(),
        primary key(aux_prod, data_hora)
    );

    delete from tbl_auxprod;

end @@
delimiter;
```

Essa primeira PROCEDURE terá a função de criar tabela temporária e limpar a mesma, a limpeza se faz necessária porque se a mesma já existir, não será criada novamente, nesse caso só deverá ser limpa.

Na procedure a seguir temos as 4 instruções básicas:

1. Declare o cursor: Nomeando-o e definindo a estrutura da consulta a ser executada dentro dele;
2. Abra o cursor: A instrução **OPEN** executa a consulta e vincula as variáveis que estiverem referenciadas. As linhas identificadas pela consulta são chamadas de conjunto ativo e estão agora disponíveis para extração;
3. Extrair dados do cursor: Após cada extração você testa o cursor para qualquer linha existente. Se não existirem mais linhas para ser processada, você precisará fechar o cursor;
4. Feche o cursor: A instrução **CLOSE** libera o conjunto ativo de linhas;

Você utiliza as instruções **OPEN**, **FETCH** e **CLOSE** para controlar um cursor. A instrução **OPEN** executa a consulta associada ao cursor, identifica o conjunto ativo e posiciona o cursor, um indicador antes da primeira linha. A instrução **FETCH** recupera a linha atual e avança o cursor para a próxima linha. Após o processamento da última linha, a instrução **CLOSE** desativa o cursor. Veja a PROCEDURE a seguir:

```
#PROCEDURE PARA AJUSTE DE ESTOQUE
delimiter $$
• create procedure ajusta_estoque (in qtde int, out msg char(100))
begin
    -- definição de variáveis utilizadas na procedure
    declare p_linha int default 0;
    declare p_codigo int default 0;
    declare p_descri varchar(100);
    declare p_estoque int default 0;
    declare p_status int default 0;

    -- definição do cursor
    declare meucursor cursor for
        select cod_produto, nome_produto, estoque_atual from tbl_produto;

    -- definição da variável de controle de looping do cursor
    declare continue handler for not found set p_linha = 1;

    -- abertura do cursor
    open meucursor;

    -- chamada da procedure de criação da tabela temporária
    call cria_tabela();

    -- looping de execução do cursor
    meuloop: loop
        fetch meucursor into p_codigo, p_descri, p_estoque;

        -- controle de existir mais registros na tabela
        if p_linha = 1 then
            select count(*) into p_status from tbl_auxprod;
            if p_status > 0 then

                -- seleciono a tabela temporária
                select * from tbl_auxprod;
                leave meuloop;
            else

                -- a procedure rodou mas sem nenhum processamento
                set msg = "nada processado!";
                select msg;
                leave meuloop;
            end if;
        elseif p_estoque = qtde then

            -- atualizo o estoque
            update tbl_produto set estoque_atual = qtde + 2
            where cod_produto = p_codigo;

            -- insere os dados na tabela temporária
            insert into tbl_auxprod (aux_prod,aux_desc, qtde_ajust)
            values (p_codigo, p_descri, p_estoque + 2);
        end if;

    end loop meuloop;

    -- fechamento do cursor
    close meucursor;
end $$
delimiter ;
```

Atividade de fixação entregar até o dia 06/12

De acordo com as aulas dadas, realizem as tarefas conforme abaixo:

1º - Analisem a procedure que acabamos de fazer e procure melhorar sua performance, otimizando-a de acordo com a proposta passada pelo professor;

2º - Modifique a procedure que usamos em cursores de forma que ao passar um parâmetro (percentual de aumento), a tabela de produto tenha o valor do mesmo reajustado, mas somente para os produtos em que o valor seja inferior a R\$ 10,00.

6. Functions (Funções)

Funções são definidas como procedimentos que retornam um valor. Elas são ativadas na avaliação das expressões.

Para criarmos a função usamos o CREATE FUNCTION, e excluirmos a função usamos o DROP FUNCTION, nem vamos perder muito tempo com isso, vamos a sintaxe de criação da Função:

```
1 DELIMITER $$
2 CREATE FUNCTION NOME_DA_FUNCAO(PARAMETRO TIPO_DADO)
3 RETURNS TIPO_RETORNO
4 DETERMINISTIC
5 BEGIN
6     #CORPO DA FUNCTION E FINALIZANDO COM O RETORNO
7     RETURN PARAMETRO_DE_RETORNO;
8 END $
```

O DETERMINISTIC é aquele que sempre gera o mesmo resultado para os mesmos parâmetros de entrada. Se o resultados gerados são diferentes, a função é considerada não determinística.

O RETURNS TIPO_RETORNO informa o tipo de dado do retorno da função, caso este dado seja de tipo diferente, ele irá ser adaptado ao formato da Function, por exemplo, o dado é do tipo Date, e você pediu um retorno CHAR, este dado Date será retornado como CHAR.

Vamos fazer alguns exemplos de acordo com o banco de dados das nossas aulas.

Neste primeiro exemplo vamos utilizar a função FORMAT() do MYSQL para formatar valores monetários, levando em consideração a notação estrangeira, passando os centavos para duas casas decimais: o divisor de casas decimais é o ponto e o de milhares, a vírgula. Para inverter a situação, vamos recriar a função incorporando o uso da função nativa REPLACE(), que serve para substituir itens existentes, como por exemplo substituir uma vírgula por um ponto.

Vamos fazer uma função que retorne os valores dos produtos em reais, por exemplo R\$ 3,50 ou R\$ 1.250,14 observem a FUNCTION abaixo:

```
delimiter $
• create function formatar_moeda(valor float) returns varchar(15)
deterministic
begin
    declare formatado varchar(15);
    set formatado = format(valor,2);
    set formatado = replace(formatado,".", "#");
    set formatado = replace(formatado,"", ".");
    set formatado = replace(formatado,"#", ",");
    return concat('R$ ', formatado);
end $
```

Vamos agora ver como executáramos tal função:

```
select valor, formatar_moeda(valor) from tbl_produto
```

Passamos a execução da função no SELECT, assim teremos o seguinte resultado:

	valor	formatar_moeda(valor)
	12.50	R\$ 12.50
	7.50	R\$ 7.50
	5.50	R\$ 5.50
	6.50	R\$ 6.50
	7.89	R\$ 7.89
	4.50	R\$ 4.50
	6.90	R\$ 6.90
	6.99	R\$ 6.99
	2.50	R\$ 2.50
	11.50	R\$ 11.50

Um outro exemplo de função seria passarmos a data no padrão brasileiro para o MySQL e mesmo assim ele nos retornaria os dados desejados, mas sabemos que só é aceito o padrão americano, mas para isso poderíamos construir uma função que nos ajudasse nesta tarefa, vejamos:

```
delimiter $
create function data_brasil(dtatual datetime) returns varchar(10)
deterministic
begin
    return date_format(dtatual, '%d/%m/%Y');
end $
```

Nesta função estamos usando a função *DATE_FORMAT*, onde a mesma formata um campo data conforme parametrizado entre parênteses, onde passamos o campo data a ser formatado, seguido pela formatação desejada, neste caso o campo vai ser formatado em dd/mm/YYYY, onde o Y maiúsculo determina que o ano será com 4 dígitos, caso queira 2 dígitos, é só passar o y em minúsculo.

Agora para testarmos vamos fazer alguns SELECTs para testar, vejamos abaixo:

```
select * from tbl_pedido
where data_brasil(data_pedido) >= '01/01/2015'
```

Neste exemplo, vejamos que passamos a execução da FUNÇÃO com o campo DATA_PEDIDO entre parênteses na cláusula *WHERE*, e na hora que informamos a data desejada, a mesma é informada no padrão brasileiro, ou seja, a nossa função que fará o MySQL entender esta data.

Um outro exemplo:

```
select *, data_brasil(data_pedido)
from tbl_pedido
where data_brasil(data_pedido) >= '01/01/2015'
```

Neste caso, iremos ver todos os campos da tabela TBL_PEDIDO e mais um campo com a função mostrando a DATA_PEDIDO formatada.

Vamos agora usar outra FUNÇÃO com cálculo, observem:

```
delimiter $
create function reajuste_valor(valor decimal(10,2), perc decimal(5,2))
returns varchar(15)
deterministic
begin
    declare vlr_reaju float;
    set vlr_reaju = valor + (valor * perc) / 100;
    return vlr_reaju;
end $
```

Nesta função passamos no *SELECT* o campo VALOR, e em seguida a FUNÇÃO REAJUSTE_VALOR, que é uma função que reajustará de acordo com o percentual passado o valor do produto, conforme mostrado abaixo:

```
select valor, reajuste_valor(valor,12.5)
from tbl_produto
```

Executando o *SELECT* teremos o seguinte resultado:

valor	reajuste_valor(valor,12.5)
12.50	14.0625
7.50	8.4375
5.50	6.1875
6.50	7.3125
7.89	8.8762502670288
4.50	5.0625
6.90	7.7624998092651
6.99	7.8637499809265
2.50	2.8125
11.50	12.9375

Podemos complementar este exemplo fazendo a execução da função FORMATAR_MOEDA dentro da *Function* REAJUSTE VALOR, para isso teremos que modificar a *Function* REAJUSTE_VALOR e acrescentando a execução da function FORMATAR_MOEDA, para isso primeiramente iremos “DROPAR” function, veja abaixo:

```
drop function reajuste_valor;
```

Modificamos a function conforme abaixo e compilamos novamente, vejamos:

```
delimiter $
create function reajuste_valor(valor decimal(10,2), perc decimal(5,2))
returns varchar(15)
deterministic
begin
    declare vlr_reaju float;
    set vlr_reaju = valor + (valor * perc) / 100;
    return formatar_moeda(vlr_reaju);
end $
```

Chamada da Função **FORMATAR_MOEDA** dentro da função REAJUSTE_VALOR

Executando novamente o *SELECT* abaixo com a mesma Função:

```
select valor, reajuste_valor(valor,12.5)
from tbl_produto
```

Agora o resultado seria assim:

valor	reajuste_valor(valor, 12.5)
12.50	R\$ 14.06
7.50	R\$ 8.44
5.50	R\$ 6.19
6.50	R\$ 7.31
7.89	R\$ 8.88
4.50	R\$ 5.06
6.90	R\$ 7.76
6.99	R\$ 7.86
2.50	R\$ 2.81
11.50	R\$ 12.94

Administração de usuários e direitos de acesso.

O MySQL permite a criação de usuários que por sua vez são associados a hosts, isso permite que se crie vários usuários inclusive com o mesmo nome desde que esses estejam em hosts diferentes.

Esses usuários são armazenados em uma base de dados dentro do SGBDR MySQL o nome da base de dados é "mysql", essa por sua vez possui varias tabelas uma dessas tabelas é a "user" que armazena os dados de cada usuário.

Observe a imagem abaixo:

Field	Type	Null	Key	Default	Extra
Host	char(60)	NO	PRI		
User	char(16)	NO	PRI		
Password	char(41)	NO			
Select_priv	enum('N','Y')	NO		N	
Insert_priv	enum('N','Y')	NO		N	
Update_priv	enum('N','Y')	NO		N	
Delete_priv	enum('N','Y')	NO		N	
Create_priv	enum('N','Y')	NO		N	
Drop_priv	enum('N','Y')	NO		N	
Reload_priv	enum('N','Y')	NO		N	
Shutdown_priv	enum('N','Y')	NO		N	
Process_priv	enum('N','Y')	NO		N	
File_priv	enum('N','Y')	NO		N	
Grant_priv	enum('N','Y')	NO		N	
References_priv	enum('N','Y')	NO		N	
Index_priv	enum('N','Y')	NO		N	
Alter_priv	enum('N','Y')	NO		N	
Show_db_priv	enum('N','Y')	NO		N	
Super_priv	enum('N','Y')	NO		N	
Create_tmp_table_priv	enum('N','Y')	NO		N	
Lock_tables_priv	enum('N','Y')	NO		N	
Execute_priv	enum('N','Y')	NO		N	
Repl_slave_priv	enum('N','Y')	NO		N	
Repl_client_priv	enum('N','Y')	NO		N	
Create_view_priv	enum('N','Y')	NO		N	
Show_view_priv	enum('N','Y')	NO		N	
Create_routine_priv	enum('N','Y')	NO		N	
Alter_routine_priv	enum('N','Y')	NO		N	
Create_user_priv	enum('N','Y')	NO		N	
Event_priv	enum('N','Y')	NO		N	
Trigger_priv	enum('N','Y')	NO		N	
Create_tablespace_priv	enum('N','Y')	NO		N	
ssl_type	enum('','ANY','X509','SPECIFIED')	NO			
ssl_cipher	blob	NO		NULL	
x509_issuer	blob	NO		NULL	
x509_subject	blob	NO		NULL	
max_questions	int(11) unsigned	NO		0	
max_updates	int(11) unsigned	NO		0	
max_connections	int(11) unsigned	NO		0	
max_user_connections	int(11) unsigned	NO		0	
plugin	char(60)	NO			
authentication_string	text	NO		NULL	

É possível observar nessa tabela que além de armazenar os dados de usuários ela também permite a configurar os direitos que o usuário vai ter sobre as bases de dados.

Para acessar essa base de dados e visualiza os usuários cadastrados basta executar os códigos abaixo:

```

1 #SELECIONA A BASE DE DADOS mysql.
2 • USE mysql;
3
4 #SELECIONA TODOS OS DADOS DE host e user DA TABELA User.
5 • SELECT host, user FROM User;

```

Criação de usuários.

Para criar novos usuários o MySQL fornece o comando “CREATE USER” sua sintaxe é a seguinte:

```
CREATE USER <nome do usuário @ host> IDENTIFIED BY <senha>
```

Observe que definição de qual host o usuário vai pertencer deve ser indicada após o símbolo de arroba “@”. A imagem abaixo ilustra como criar dois usuários o primeiro com acesso a todos os hosts o segundo com acesso apenas ao “localhost”, e ambos identificados pela senha “teste”:

```

1 • use mysql;
2
3 #CRIAÇÃO DE USUÁRIO COM ACESSO A TODOS OS HOSTS DO MYSQL
4 • CREATE USER 'aula_usu01'@'%' IDENTIFIED BY 'teste';
5
6 #CRIAÇÃO DE USUÁRIO COM ACESSO APENAS AO LOCALHOST
7 • CREATE USER 'aula_usu02'@'localhost' IDENTIFIED BY 'teste';

```

Se executarmos uma “select” na tabela “user” da base de dados “mysql” teremos um resultado muito parecido com a imagem abaixo:

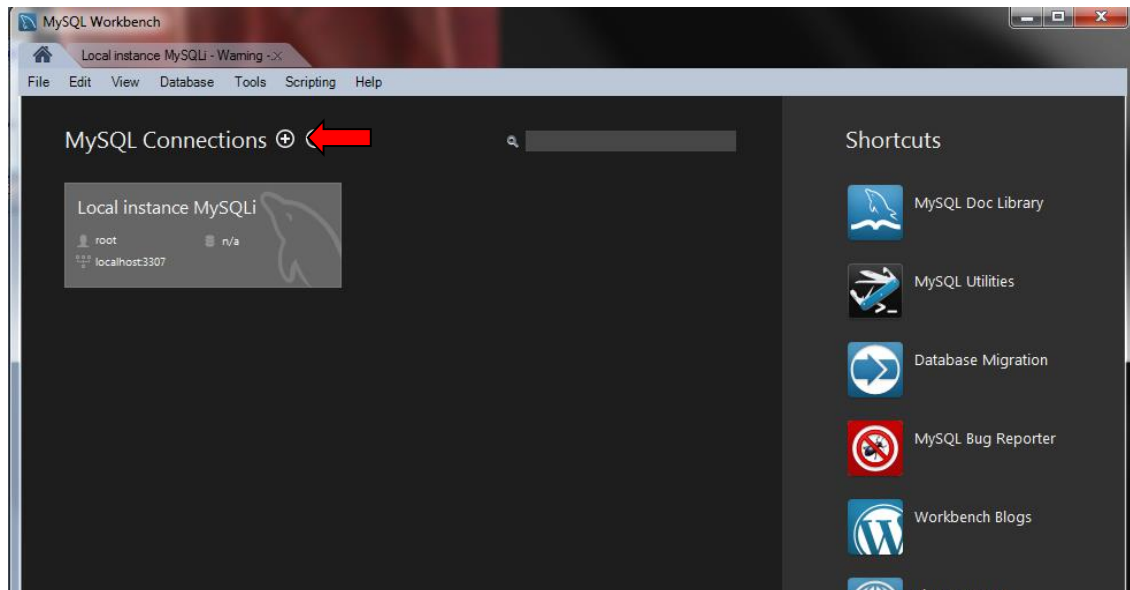
9 • `select user, password, host from User order by user asc;`

user	password	host
aula_usu01	*A00D6EEF76EC509DB66358D2E6685F8FF7A4C3DD	%
aula_usu02	*A00D6EEF76EC509DB66358D2E6685F8FF7A4C3DD	localhost
pma		localhost
root		localhost
root		linux

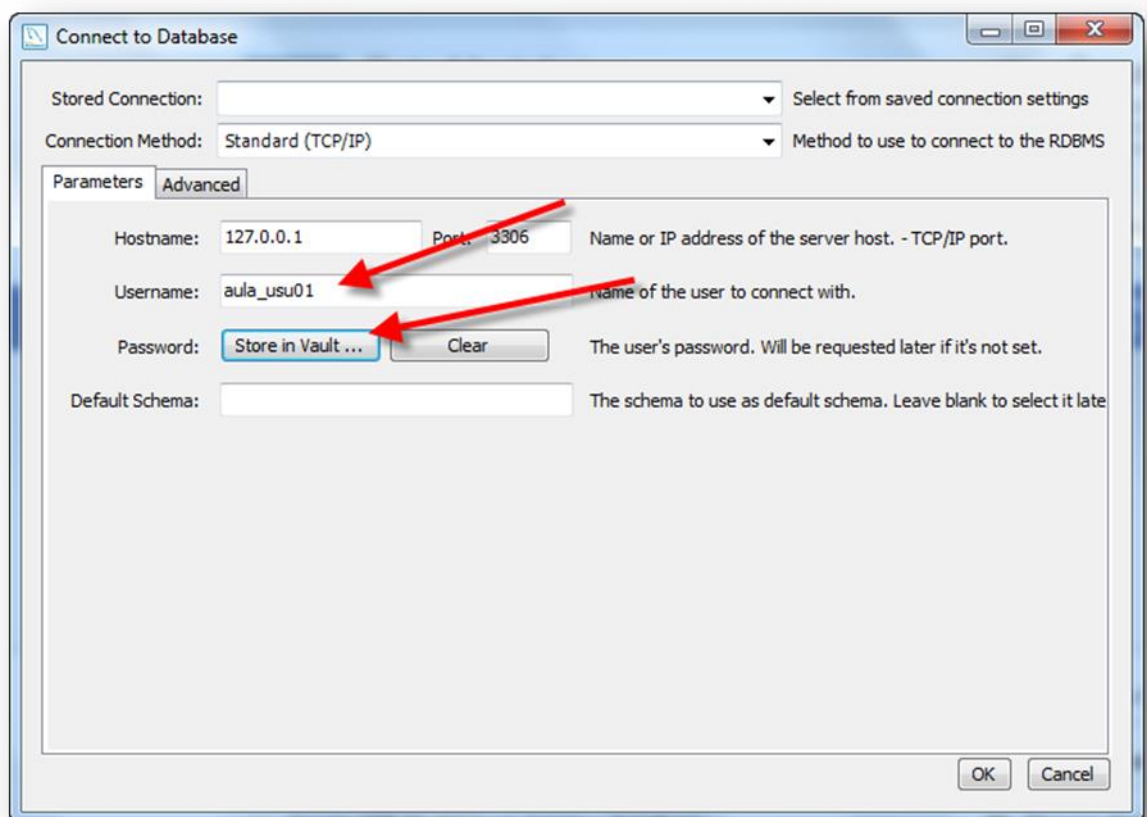
Percebam que os usuários foram devidamente criados de acordo com o hosts informados e a senha está criptografada no padrão MD5.

Logando com um usuário criado.

No Mysql WorkBench clique no botão de “home” a “casinha” no canto superior esquerdo e na coluna clique em “Open Connection to Start Queryng”, observe a imagem abaixo:



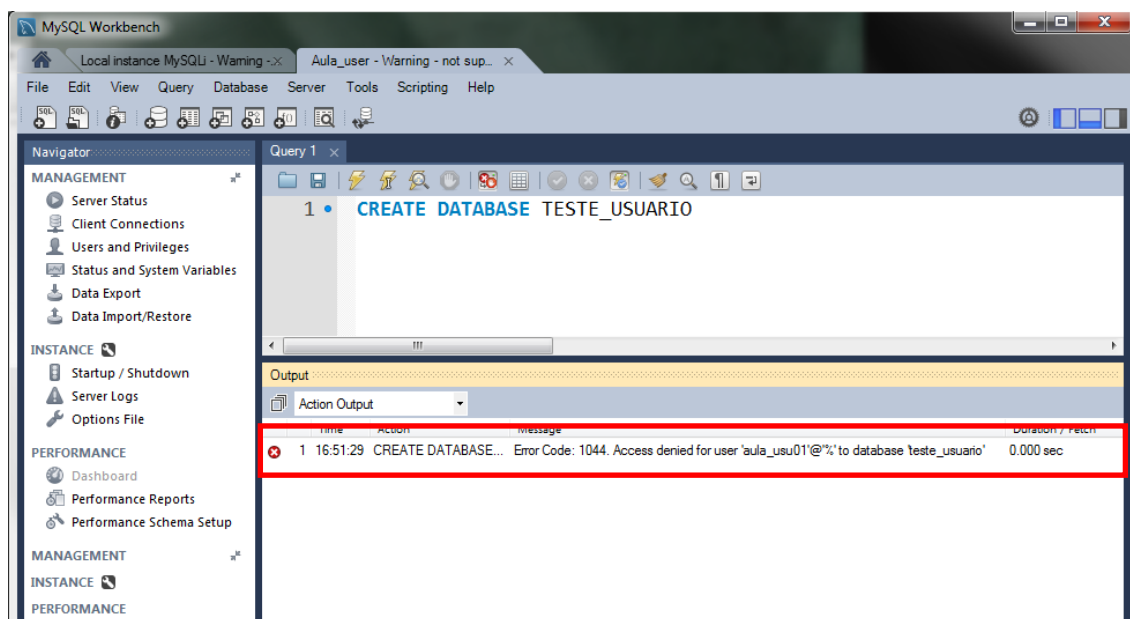
Uma nova janela abrirá onde poderemos informar um novo nome de usuário e uma nova senha, observe a imagem abaixo:



Após informar o nome do usuário clique sobre o botão “Store in Vault” para informar a senha, observe a imagem abaixo:



Note que a não existirá uma única base de dados disponível para o novo usuário então podemos começar a criar novas bases e tabelas com esse usuário dentro do host em questão? Ainda não, para que esse novo usuário “enxergue” bases de dados que já existem e que foram criadas por outros usuários devemos definir permissões. Olhe o teste que fizemos para comprovar isto:



Percebam que ao tentarmos criar um novo banco de dados, o mesmo informa que não temos direito para tal operação.

Permissões de usuários.

O comando mysql para adicionar permissões a novos usuários é o “GRANT” sua sintaxe é:

```
GRANT privileges ( column_list )
ON [ object_type ] privilege_level
TO account [ IDENTIFIED BY 'password' ]
[ REQUIRE encryption ]
WITH with_options
```

Onde:

privileges indica os privilégios que você atribui à conta. Por exemplo, o **CREATE** permite que um usuário possa criar bancos de dados e tabelas. Você pode conceder vários privilégios usando único **GRANT** separando os por vírgulas.

column_list especifica as colunas onde um privilégio se aplica. As colunas são separadas por vírgulas e listada entre parênteses. O **column_list** é um elemento opcional.

privilege_level especifica o nível em que os privilégios se aplicam. Você pode usar os privilégios globais, os privilégios específicos de banco de dados, os privilégios específicos de tabela, os privilégios específicos de coluna, etc.

account especifica a qual conta está sendo concedido os privilégios.

password especifica a senha para atribuir à conta. Se a conta existir, o **GRANT** substitui a senha antiga por uma nova. Como no **CREATE USER** você usa senhas em texto puro seguido pelo **IDENTIFIED BY**. O **IDENTIFIED BY** é opcional.

Após a **REQUIRE**, você especifica se a conta tem que se conectar ao servidor de banco de dados através de uma conexão segura usando **SSL**.

Se você quiser que a conta possa conceder seus privilégios para outras contas, você precisa usar o **WITH** com **GRANT OPTION**. Além disso, você pode usar o **WITH** para alocar recursos, por exemplo, para definir quantas conexões ou declarações que uma conta pode usar por hora.

Lista de privilégios:

PRIVILÉGIO	DESCRIÇÃO
ALL [PRIVILEGES]	Conceder todos os privilégios no nível de acesso especificado, exceto GRANT OPTION
ALTER	Permitir a utilização de ALTER TABLE
ALTER ROUTINE	Permitir ao usuário alterar ou descartar rotina armazenada
CREATE	Permitir que o usuário criar banco de dados e a tabela
CREATE ROUTINE	Permitir que o usuário crie rotina armazenada
CREATE TABLESPACE	Permitir que o usuário criar, alterar ou descartar tablespaces e log grupos de arquivos
CREATE TEMPORARY TABLES	Permitir que o usuário para criar a tabela temporária usando CREATE TABLE TEMPORÁRIA
CREATE USER	Permitir que o usuário utilizar o USER CREATE, DROP USER, RENAME USER, e REVOKE ALL declarações privilégios.
CREATE VIEW	Permitir que o usuário criar ou modificar vista
DELETE	Permitir que o usuário utilize DELETE
DROP	Permitir que o usuário dropar banco de dados, tabela e exibição
EVENT	Permitir que o usuário agendar eventos no Event Scheduler
EXECUTE	Permitir que o usuário execute stored procedures
FILE	Permitir que o usuário ler qualquer arquivo no diretório de banco de dados.
GRANT OPTION	Permitir que o usuário tem privilégios para conceder ou revogar privilégios de outras contas
INDEX	Permitir que o usuário criar ou remover índices.
INSERT	Permitir que o usuário use INSERT
LOCK TABLES	Permitir que o usuário usar LOCK TABLES em tabelas nas quais você tem o privilégio SELECT
PROCESS	Permitir que o usuário ver todos os processos com instrução SHOW PROCESSLIST.
PROXY	Ativar proxy usuário
REFERENCES	Não implementado
RELOAD	Permitir que o usuário usar operações FLUSH

PRIVILÉGIO	DESCRIÇÃO
REPLICATION CLIENT	Permitir ao usuário consultar para ver onde servidores mestres ou escravos são
REPLICATION SLAVE	Permitir que o usuário usar escravos replicadas para ler eventos do log binário do mestre.
SELECT	Permitir que o usuário use instrução SELECT
SHOW DATABASES	Permitir que o usuário para mostrar todos os bancos de dados
SHOW VIEW	Permitir que o usuário usar SHOW CREATE VIEW
SHUTDOWN	Permitir que o usuário use mysqladmin comando de desligamento
SUPER	Permitir que o usuário utilizar outras operações administrativas, como CHANGE MASTER TO, MATAR, PURGE BINARY LOGS, SET GLOBAL, e comando mysqladmin
TRIGGER	Permitir que o usuário usar operações gatilho.
UPDATE	Permitir que o usuário use instrução UPDATE
USAGE	Equivalente a "sem privilégios"

Para exemplificar melhor a nossa aula vamos dar direitos de SELEÇÃO ao usuário: aula_usu01, conforme abaixo:

12 • **GRANT SELECT ON BD_VENDAS.TBL_CLIENTE TO aula_usu01;**

A ideia é que este usuário somente faça *SELECTs* na tabela TBL_CLIENTE do banco de dados BD_VENDAS,

The screenshot shows the MySQL Workbench interface. On the left, the 'SCHEMAS' pane shows the database structure, including the 'bd_vendas' database and the 'tbl_cliente' table. The 'Query 1' window contains the following SQL script:

```

1 • use bd_vendas;
2
3 • drop table tbl_cliente;
4
5 • select * from tbl_log;
6
7 • select * from tbl_cliente
8

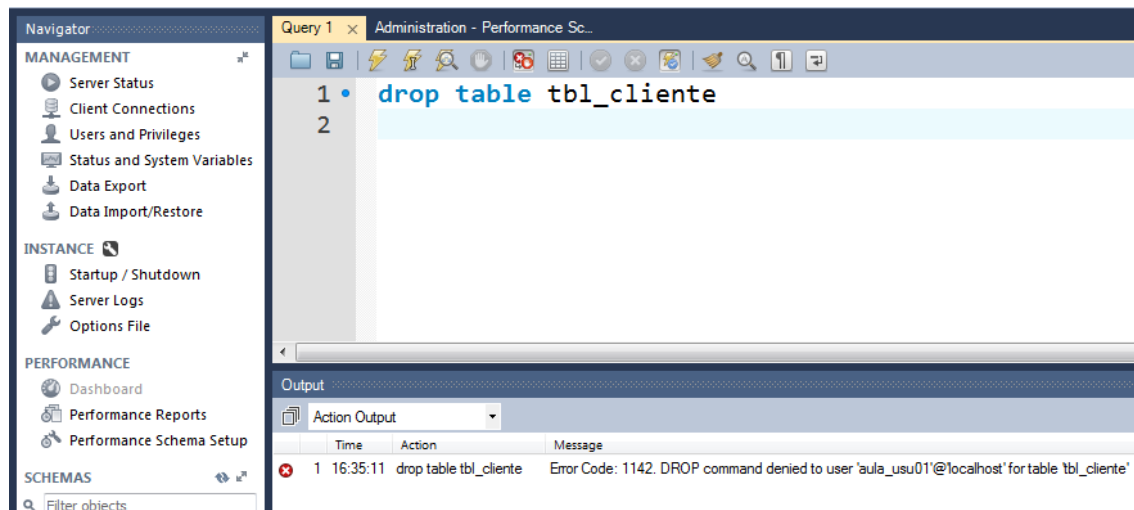
```

The 'Result Grid' shows the results of the query, displaying a table with two columns: 'cod_cliente' and 'nome_cliente'. The results are as follows:

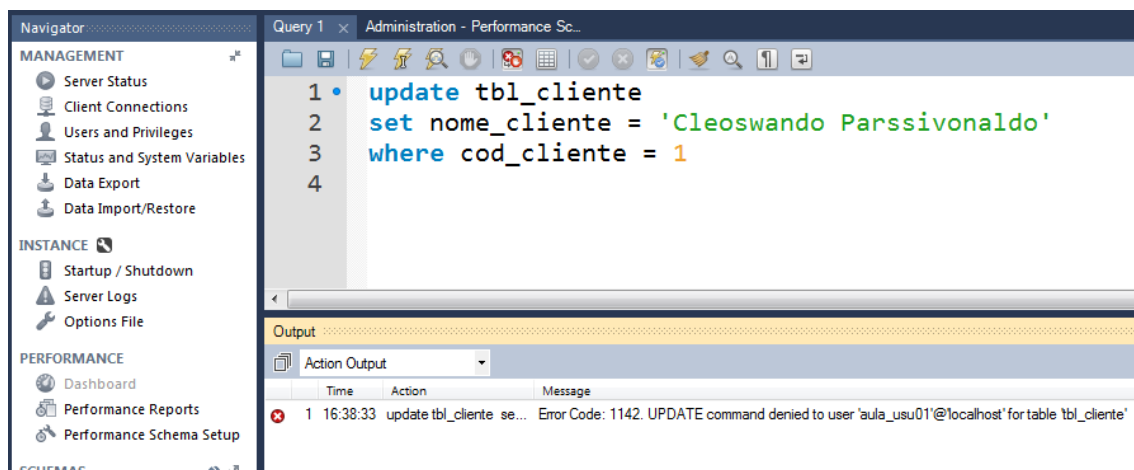
cod_cliente	nome_cliente
1	MARCOS COSTA
2	JOAQUIM SANTOS
3	ELIANE PRADO
4	ARTHUR PRADO COSTA
5	DANILO CAMPOS

The 'Output' window at the bottom shows the execution of the query, with a message indicating that 5 row(s) returned.

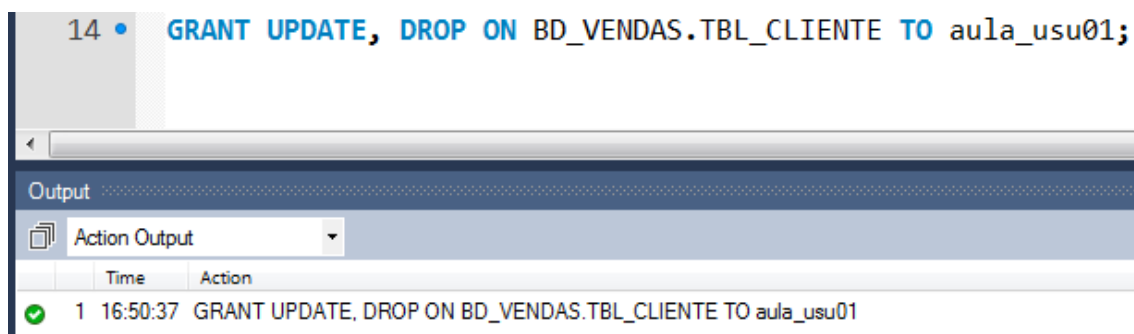
Se usarmos outro comando não habilitado como um DROP ou UPDATE, não teremos permissão, vejamos:



E



Liberando o acesso os procedimentos anteriores serão permitidos



Agora os comandos anteriores funcionam normalmente

```
1 • update tbl_cliente
2   set nome_cliente = 'Cleoswando Parssivonaldo'
3   where cod_cliente = 1
4
```

Output			
Action Output			
	Time	Action	Message
✓	1 16:52:32	update tbl_cliente se...	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0

Só não irei “Dropar” a minha tabela é claro!
Revogando direitos de acesso.

Da mesma forma que podemos dar direitos a usuários dentro de uma base de dados podemos remover tais direito, para tanto se usa o comando “**revoke**”, pois é muito parecida com o comando GRANT, vejamos:

```
16 • REVOKE SELECT ON BD_VENDAS.TBL_CLIENTE FROM aula_usu01;
17
```

Output			
Action Output			
	Time	Action	Message
✓	1 17:01:52	REVOKE SELECT ON BD_VENDAS.TBL_CLIENTE FROM aula_usu01	

Neste caso o usuário não poderá mais fazer SELECT na tbl_cliente

```
6 • select * from tbl_cliente
```

Output			
Action Output			
	Time	Action	Message
✗	1 17:03:27	select * from tbl_cliente	Error Code: 1142. SELECT command denied to user 'aula_usu01'@'localhost' for table 'tbl_cliente'

Exercícios:

Criem mais dois usuários para o Banco Bd_Vendas, e logo depois concedam acessos que achem necessários e depois revoguem os mesmos.

1. Introdução ao NoSQL.

Em nosso curso de Banco de Dados III iremos abordar um tema que está em alta em desenvolvimento de sistemas que é a utilização de banco de dados NoSQL, tema essa que reflete uma boa parte das vagas no mercado de trabalho, é um conteúdo a mais no curso onde a grade do curso de desenvolvimento de sistemas não aborda essa temática.

1.1 Adventos e tecnologia NoSQL.

Banco de dados NoSQL (*Not Only SQL*) é um assunto em alta. O termo banco de dados NoSQL tornou-se uma parte permanente da nomenclatura de armazenamento de dados, usado para descrever esquemas de armazenamento de dados não-relacionais e sem esquemas.

A *Structured Query Language* (Linguagem SQL) tornou-se o padrão para manipulação de dados em sistemas de gerenciamento de banco de dados relacionais ao longo dos anos. Por conta disso, o termo NoSQL vem sendo usado para definir tudo que não seja relacional, embora nem sempre essa abordagem seja precisa.

Sendo preciso ou não o termo NoSQL, as várias tecnologias NoSQL são bem-vindas e necessárias as bases de dados de hoje, especialmente por conta do conceito de Big Data.

Os bancos de dados NoSQL surgiram como uma solução para a questão da escalabilidade no armazenamento e processamento de grandes volumes de dados na Web 2.0, em 2009, quando algumas novas empresas da Web 2.0 e a comunidade de software livre e código aberto começaram a desenvolver novas opções de bancos de dados. O termo NoSQL tem sido usado com o significado de “Não apenas SQL” numa tentativa da comunidade de reconhecer a utilidade dos modelos tradicionais e não divergir as discussões. Entre os principais fatores que favoreceram seu surgimento estão a natureza dos dados da web, a importância de se atingir altos graus de paralelismo no processamento de dos grandes volumes de dados e a distribuição de sistemas em escala global.

Se hoje você produz dados usando o seu celular, no futuro você gerará dados andando dentro da sua casa. Não é exagero, IoT é uma realidade em amadurecimento.

“Tá, mas o que o banco de dados NoSQL tem a ver com isso tudo?”

Na realidade, tudo. Big data é a matéria prima dos bancos de dados NoSQL e, se você quer se aprofundar em Data Science, Engenharia de Dados, BI, entre tantas outras áreas de TI, não vai conseguir sem conhecer NoSQL, o termo “NoSQL” surgiu em 1998, mas foi em 2006 – quando foi citado pelo Google – que o termo se popularizou. Era uma época onde os bancos relacionais não mais suportavam a massa de dados da internet. Só a internet hoje armazena alguns terabytes de dados.

1.2 Banco de Dados NoSQL: como funciona?

O termo “NoSQL” é utilizado para designar os bancos de dados não relacionais e quase sempre é relacionado com Big Data.

Isso porque, o conceito de Big Data está em ascensão e é matéria prima dos bancos de dados NoSQL. Para entender os NoSQLs, é importante saber que a linguagem SQL sempre foi usada para tratamento de dados em bancos relacionais ao longo dos anos.

Mas por que NoSQL?

Parece que estamos falando de bancos que invariavelmente não podem ser tratados com a linguagem SQL, quando na verdade não é bem assim.

Na verdade, a nomenclatura NoSQL é só para fazer uma diferenciação entre bancos reconhecidamente relacionais, como MySQL, PostgreSQL e etc.

Em suma, a principal diferença entre os bancos de dados relacionais e NoSQL é que o segundo permite maior velocidade, flexibilidade e escalabilidade ao armazenar e acessar dados não estruturados.

1.3 Alguns bancos de dados NoSQL

1.3.1 MongoDB

Quando falamos em MongoDB estamos falando de um líder de mercado dos bancos de dados NoSQL. Para armazenar dados, o MongoDB utiliza alguns documentos muito similares ao formato JSON. O melhor de tudo – e talvez a razão de ser líder de mercado – é que o MongoDB é open source, o que contribui muito para a evolução da sua tecnologia.

1.3.2 Amazon DynamoDB

Mais um produto excelente da AWS (Amazon Web Services). O banco de dados DynamoDB é totalmente cloud e viabiliza um desempenho confiável e em escala.

Um ponto bem importante: a Amazon confirma que a latência é consistente e fica abaixo de 10 milissegundos. Além disso, tem recursos valiosos de segurança, baseados em cache de memória, backup e restauração de dados.

Este banco de dados já é amplamente utilizado, assim como o MongoDB, e pode ser utilizado para criação de datastore, jogos, ad tech e aplicativos web sem servidor.

1.3.3 Cassandra

Muitas pessoas não sabem, mas o Cassandra foi desenvolvido no Facebook. Hoje em dia, o Cassandra – assim como o HBase – são mantidos pela Apache Foundation.

Isso até faz sentido, considerando a quantidade de dados que a rede social gera a cada milissegundo. Mas, afinal, por que Cassandra é tão popular para trabalhar com Big Data? O fato é que Cassandra é muito otimizado para clusters, especialmente por funcionar sem mestres. O fato de ter mecanismos distribuídos também otimiza bastante a operação com os clusters. Um outro ponto forte do Cassandra é o conceito de orientação por coluna, o que torna a latência bem menor em algumas pesquisas.

1.3.4 Redis

O Redis é um modelo de armazenamento de dados, que é open source e foi lançado em 2009. Os dados são armazenados na forma de chave-valor e na memória do Redis, o que o torna rápido e flexível. Trata-se do banco NoSQL mais famoso do tipo chave-valor.

Assim como os dois primeiros, o Redis possui baixíssima latência. O Redis também é fácil de usar e muito rápido.

1.3.5 HBase

O HBase é um banco de dados open source, orientado a colunas e distribuído. Atualmente, Spotify e Facebook são algumas das grandes corporações que utilizam esse modelo de armazenamento.

O HBase foi formatado a partir do BigTable do Google e também é escrito em Java. É justamente por isso que tem fácil integração com o MapReduce.

Pra quem não sabe, o MapReduce é uma ferramenta do framework Apache Hadoop, uma das principais plataformas para tratamento de big data.

1.4 NoSQL vs SQL

SQL	NoSQL
Armazenamento de Dados Estruturados por Tabela	Armazenamento de Dados estruturados e não-estruturados por colunas, grafos, chave-valor e documentos.
Esquema estático	Esquema dinâmico
Maturidade de suporte maior (geralmente pago)	Suporte por comunidade independente (open source)
Escalabilidade vertical	Escalabilidade horizontal
Pago	Gratuito
O desempenho não é alto em todas as consultas. Não suporta pesquisas e cruzamentos muito complexos.	Alto desempenho em consultas
Necessidade de predefinição de um esquema de tabela antes da adição de qualquer dado	Altamente flexível (fácil adição de colunas e campos de dados não estruturados)

Quadro NoSQL x SQL

Podemos assim concluir que um banco de dados NoSQL é extremamente útil quando o assunto é grande volume de dados. Se estivermos falando de uma corporação pequena, que

não aprofunda tanto assim a análise e tratamento de dados, o banco relacional funciona muito bem.

E não, o banco de dados NoSQL não veio para substituir o relacional. Veio para ser uma alternativa, em meio a um mundo onde pouquíssimos dados ainda são usados para inteligência.

1.5. Seminário de BD III

Dia 14 de Agosto de 2019, em grupos, elaborem um seminário abordando as seguintes temáticas:

- 1 – MongoDB;
- 2 – Amazon DynamoDB
- 3 – Cassandra
- 4 – Redis
- 5 – Hbase

Cada grupo terá 15 minutos para apresentação, onde irei considerar como a 1º menção de Banco de Dados III.

Result Grid										
		Filter Rows:		Export:		Wrap Cell Content:				
	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	tbl_cliente	ALL	NULL	NULL	NULL	NULL	10365	Using where