

# MySQL

## Readme.rmd

Sergio Pedro R Oliveira

2022-06-04

## Contents

<b>1</b>	<b>Objetivo</b>	<b>3</b>
<b>2</b>	<b>Referência</b>	<b>3</b>
<b>3</b>	<b>Módulo 2 - Teoria</b>	<b>4</b>
3.1	Modelagem . . . . .	4
3.2	Tipagem de campos . . . . .	7
3.3	Subtipos - regras e restrições . . . . .	9
<b>4</b>	<b>Módulo 3 - Comandos</b>	<b>11</b>
4.1	Inserir registros na tabela - <b>INSERT</b> . . . . .	11
4.2	Consultando campos na tabela - <b>SELECT</b> . . . . .	12
4.3	Consultando registros na tabela - <b>WHERE</b> . . . . .	13
<b>5</b>	<b>Módulo 5 - Operadores Lógicos, GROUP BY e ORDER BY</b>	<b>14</b>
5.1	Operadores Lógicos e Performance de operadores lógicos . . . . .	14
5.2	Agregador e funções de agregação - <b>GROUP BY</b> . . . . .	16
5.3	Ordenando registros - <b>ORDER BY</b> . . . . .	17
<b>6</b>	<b>Módulo 7 - Mais comandos UPDATE e DELETE</b>	<b>18</b>
6.1	Atualizando registros na tabela - <b>UPDATE</b> . . . . .	18
6.2	Deletando registros - <b>DELETE</b> . . . . .	19
6.3	Transação - <b>START TRANSACTION</b> . . . . .	20
<b>7</b>	<b>Módulo 8 - Modelagem</b>	<b>21</b>
7.1	Primeira forma normal . . . . .	21
7.2	Segunda forma normal . . . . .	22
7.3	Terceira forma normal . . . . .	23
<b>8</b>	<b>Módulo 9 - PROJEÇÃO, SELEÇÃO E JUNÇÃO</b>	<b>24</b>
8.1	PROJEÇÃO . . . . .	24
8.2	SELEÇÃO . . . . .	24
8.3	JUNÇÃO . . . . .	24
<b>9</b>	<b>Categoria de comandos</b>	<b>27</b>
9.1	<b>DML</b> - <i>Data Manipulation Language</i> (Linguagem de Manipulação de Dados) . . . . .	27
9.2	<b>DDL</b> - <i>Data Definition Language</i> (Linguagem de definição de dados) . . . . .	28
9.3	<b>DCL</b> - <i>Data Control Language</i> (Linguagem de Controle de Dados) . . . . .	31
9.4	<b>TCL</b> - <i>Tool Command Language</i> (Linguagem de Comandos de Ferramentas) . . . . .	33

<b>10 Módulo 11 - Funções e VIEWS</b>	<b>34</b>
10.1 Funções . . . . .	34
10.2 VIEWS . . . . .	36
<b>11 Módulo 12 - Diagrama ER - brModelo e StarUML</b>	<b>38</b>
11.1 Peter Chen . . . . .	39
11.2 Cross Foot (pé de galinha) . . . . .	40
<b>12 Módulo 13 - DELIMITER e STORED PROCEDURES</b>	<b>41</b>
12.1 Como mudar o delimitador . . . . .	41
12.2 STORED PROCEDURES - Procedimentos Armazenados - Funções . . . . .	42
<b>13 Módulo 14 - Funções Básicas</b>	<b>44</b>
<b>14 Módulo 15 - Subqueries (Subconsulta) e Trabalhando com linhas</b>	<b>45</b>
14.1 Subqueries (Subconsulta) . . . . .	45
14.2 Trabalhando com linhas . . . . .	45
<b>15 Módulo 16 - Modificação de tabelas</b>	<b>46</b>
15.1 Modificação de tabelas - ALTER . . . . .	46
15.2 Constraints - regras e boas praticas . . . . .	49
15.3 Dicionario de dados do sistema . . . . .	50
<b>16 Módulo 18 - Entidade Associativa e Chaves</b>	<b>51</b>
16.1 Entidades Associativas . . . . .	51
16.2 Sobre Chaves . . . . .	52
<b>17 Módulo 19 - TRIGGERS (Gatilhos)</b>	<b>53</b>
17.1 TRIGGERS . . . . .	53
17.2 Deletando o TRIGGER . . . . .	53
17.3 Conceito de NEW e OLD . . . . .	53
17.4 Observações TRIGGER . . . . .	54
17.5 Uso de TRIGGER para BACKUP . . . . .	55
17.6 Comunicação entre bancos de dado . . . . .	56
<b>18 Módulo 20 - Autorelacionamento</b>	<b>57</b>
18.1 Autorelacionamento . . . . .	57
18.2 Como construir autorelacionamento . . . . .	57
18.3 Retornar valor relacionado no autorelacionamento . . . . .	58
<b>19 Módulo 21 - Cursores</b>	<b>59</b>
19.1 Teoria . . . . .	59
19.2 Principais palavras chaves . . . . .	60
19.3 Juntando tudo - CURSOR . . . . .	65
<b>20 Módulo 23 - Introdução a Business Intelligence</b>	<b>67</b>
20.1 Banco de dados relacional . . . . .	67
20.2 Business Intelligence . . . . .	67
<b>21 Detalhes</b>	<b>69</b>
<b>22 Andamento dos Estudos</b>	<b>70</b>
22.1 Assunto em andamento: . . . . .	70

## **1 Objetivo**

Estudo dirigido de MySQL.

## **2 Referência**

Vídeo aulas “O curso completo de Banco de Dados e SQL, sem mistérios” - Udemey.

## 3 Módulo 2 - Teoria

### 3.1 Modelagem

Obs.: alguns softwares (ex.: brModelo) chamam a modelagem lógica de modelo conceitual.

#### 1. Analise de requisitos

- Modelo das necessidades do Cliente, o que é do interesse do cliente e o que ele precisa no banco de dados.
- Processos a serem controlados pelo sistema.
- É uma fase de muita conversa e reunião com o cliente para investigar as regras do negocio.

#### 2. processos de modelagem

- Fases 01 e 02 do projeto de banco de dados são feitos pelo administrador de dados:

##### i. Modelo conceitual

- Rascunho dos requisitos do projeto.
- Desenho conceitual.

##### ii. Modelo lógico

- Coloca os requisitos num programa de diagramas.
- Cria **entidades**, posteriormente serão tabelas.
- Cria **atributos**, posteriormente serão campos, colunas nas tabelas.
- **Atributos identificador**, posteriormente será **Chave Primaria Artificial**.
  - \* Normalmente leva o nome “ID” + “o\_nome\_da\_tabela”.
- Modelo **entidades-relacionamentos**, define os relacionamentos entre os agentes.

##### \* Relacionamentos:

##### · Obrigatoriedade

A obrigatoriedade de preencher as duas tabelas/entidades.

Tipos:

0

Não existe obrigatoriedade, se entrar com dados em um, não é obrigado a entrar com dados no outro.

1

Existe obrigatoriedade, se entrar com dados em um, obrigatoriamente é necessario entrar com dados no outro.

- Cardinalidade  
Maximo de preenchimentos:  
Se obrigatoriedade 0, no minimo 0 e no maximo n dados.  
Se obrigatoriedade 1, no minimo 1 e no maximo n dados.

\* tipos de relacionamentos de entidade:

- (1,1)  
É obrigatorio, pode entrar apenas com 1 dado.
- (0,1)  
Não é obrigatorio, quando entrar, entrar com 1 dado.
- (1,n)  
É obrigatorio, pode entrar com varios dados.
- (0,n)  
Não é obrigatorio, pode entrar com varios dados.

\* Como ler os relacionamentos entre entidades:

Exemplos:

- (1,1) -> (0,n)  
Ignorar a primeira coordenanda de obrigatoriedade dos dois relacionamentos, e fica 1 para n, logo “um para muitos”.
- (0,n) -> (0,1)  
Ignorar a primeira coordenanda de obrigatoriedade dos dois relacionamentos, e fica n para 1, logo “muitos para um”.

- Fase 03 do projeto de banco de dados é feita tanto pelo administrador de bancos de dados(DBA) quanto administrador de dados(AD):

### iii. Modelo físico

- Criando banco de dados.

**CREATE DATABASE** *nome\_do\_banco\_de\_dados*;

- Conectando-se a um dos banco de dados do sistema.

**USE** *nome\_do\_banco\_de\_dados*;

- Criando tabela.

**CREATE TABLE** *nome\_da\_tabela*(  
*coluna1 tipo(tamanho) chave\_ou\_não restrições*,  
*coluna2 tipo(tamanho) restrições*,  
 ...,  
**FOREIGN KEY**(*nome\_da\_coluna\_da\_chave\_estrangeira*)  
**REFERENCES** *nome\_da\_tabela\_da\_chave\_primaria*(*nome\_da\_coluna\_da\_chave\_primaria*)  
 );

- Criando VIEWS.  
**CREATE VIEW** *VW\_nome\_da\_view* **AS**  
**SELECT**  
 ...  
**FROM** *nome\_tabela*  
 ...;
- Verificando os banco de dados no sistema.  
**SHOW DATABASES;**
- Verificando as tabelas (e **VIEWS**) do banco de dados.  
**SHOW TABLES;**
- Visualização detalhada de tabelas, mais detalhado que **DESC**.  
**SHOW CREATE TABLE** *nome\_da\_tabela*;
- *Descrevendo* como é a estrutura de uma tabela, verificando quais são as colunas.  
**DESC** *nome\_da\_tabela*;
- Verificar em qual **DATABASE** esta conectado no momento e outros *status* em uso.  
**STATUS**  
 Não precisa de “;” (delimitador) pois não é um comando **SQL**, é um comando de infraestrutura.
- Deletando um banco de dados.  
**DROP DATABASE** *nome\_do\_banco\_de\_dados*;
- Deletando uma tabela.  
**DROP TABLE** *nome\_da\_tabela*;
- Deletando uma **VIEW**.  
**DROP VIEW** *VW\_nome\_da\_view*;
- Deletando um **TRIGGER**.  
**DROP TRIGGER** *nome\_do\_trigger*;

## 3.2 Tipagem de campos

A tipagem correta diminui o tempo de resposta, otimiza os processos.

### 1. Tipo caracteres

- **CHAR**

- Usado quando o numero de caracteres não varia, separa na memoria um espaço determinado para ser preenchido.

- Sintaxe:

**CHAR**(*numero\_maximo\_de\_caracteres*)

- **VARCHAR**

- Usado quando o numero de caracteres varia, dependendo da entrada adapta o espaço separado na memoria para caber os characters.

- Sintaxe:

**VARCHAR**(*numero\_maximo\_de\_caracteres*)

### 2. Tipo **ENUM**

- Conjunto de dados enumerados, ou seja, um conjunto fixo de dados.

- Limita dados em uma coluna, lista de opções.

- tipo característico do **MySQL**.

- Sintaxe:

**ENUM**(*‘primeira\_opção’; ‘segunda\_opção’; ...*)

### 3. Tipo numerico

- **INT**

- Para numeros inteiros.

- Numero maximo de 11 digitos, para numeros maiores que isso usar **VARCHAR**.

- Sintaxe:

**INT**

- **FLOAT**

- Ponto flutuante, ou seja, numeros reais.

- Ao entrar com o valor (em **INSERT**, **UPDATE**, ...), usar “.” ao inves de “,” para separar as casas decimais.

- Para numeros com casas decimais.  
**FLOAT**(*total*, *virgula*)

#### 4. tipo data e hora

- **DATE**

- Para datas, no fomato “aaaa-mm-dd”.

- **TIME**

- Para tempo(horas), no fomato “hh:mm:ss”.

- **DATETIME**

- Para data e tempo(horas), no fomato “aaaa-mm-dd hh:mm:ss”.

- **YEAR**[(2|4)]

- Ano nos formatos de 2 ou 4 dígitos.

#### 5. Para fotos e documentos

- **BLOB**

#### 6. Tipo textos

- **TEXT**



### 3.3 Subtipos - regras e restrições

#### 3.3.1 Restrições

- **PRIMARY KEY**
  - Define que a coluna/campo é uma *Chave Primaria*.
  - *Chave Primaria* é um campo que identifique todo registro como sendo único.
- **UNIQUE**
  - Define aquela coluna/campo sem repetições.
  - Tem valores unicos.
- **NOT NULL**
  - A coluna/campo não aceita valor NULL, deve ser preenchida.
- **AUTO\_INCREMENT**
  - A coluna/campo se auto preenche com um valor inteiro não repetido, a cada registro.

### 3.3.2 Regras chave estrangeira

- **FOREIGN KEY**

- *Chave Estrangeira* é a *Chave Primaria* de uma tabela, que vai ate a outra tabela, para fazer referencia entre registros.
- Regra de onde fica a *Chave Estrangeira* (**FK**):
  - \* 1 x 1 (um pra um) a *Chave Estrangeira* fica na tabela mais fraca.
    - Se for 1 x 1, leva **UNIQUE**.
  - \* 1 x n (um pra muitos) a *Chave Estrangeira* fica na tabela n.
  - \* n x n (muitos pra muitos), necessidade da criação de uma tabela associativa (ver **capitulo 16 - Módulo 18 - Entidades Associativas e Chaves**).
    - Uma tabela associativa representa uma entidade que não existe por si só e sua existência está condicionada à existência de duas ou mais entidades com relacionamento do tipo N:N.
    - Além disso, o identificador negocial da tabela é formado exclusivamente pelas colunas que são geradas pela FK dessas tabelas relacionadas.
- Sintaxe:  
**FOREIGN KEY**(*nome\_da\_coluna\_da\_chave\_estrangeira*)

- **REFERENCES**

- Aponta para onde a *Chave Estrangeira* faz referencia, qual *Chave Primaria*.
- Sintaxe:  
**REFERENCES** *nome\_da\_tabela\_da\_chave\_primaria*(*nome\_da\_coluna\_da\_chave\_primaria*)

Obs.: A sintaxe para inserção de *Chave Estrangeira* em **MySQL** fica:

**FOREIGN KEY**(*nome\_da\_coluna\_da\_chave\_estrangeira*)

**REFERENCES** *nome\_da\_tabela\_da\_chave\_primaria*(*nome\_da\_coluna\_da\_chave\_primaria*)

Sem virgula entre eles.

## 4 Módulo 3 - Comandos

### 4.1 Inserir registros na tabela - INSERT

- Existem diversas formas de inserir dados na tabela, entre eles temos:
  - Omitindo colunas/campos.
    - \* Determina apenas a tabela, que puxa todos os campos para serem preenchidos, na ordem que aparece na tabela.
    - \* Sintaxe:  
**INSERT INTO** *nome\_da\_tabela*  
**VALUES** (*valor\_na\_coluna\_1*, *valor\_na\_coluna\_2*,...);
  - Colocando as colunas.
    - \* Especifica a ordem das entradas e os campos a serem preenchidos.
    - \* Sintaxe:  
**INSERT INTO** *nome\_da\_tabela*(*coluna\_3*, *coluna\_1*, *coluna\_2*,...) **VALUES** (*valor\_na\_coluna\_3*, *valor\_na\_coluna\_1*,...);
  - INSERT COMPACTO, somente **MySQL**.
    - \* Insere diversos registros de uma vez, na ordem que aparecem na tabela.
    - \* Sintaxe:  
**INSERT INTO** *nome\_da\_tabela*  
**VALUES** (*valor\_na\_coluna\_1\_registro1*, *valor\_na\_coluna\_2\_registro1*,...),  
(*valor\_na\_coluna\_1\_registro2*, *valor\_na\_coluna\_2\_registro2*,...),  
...;
  - Inserindo dados num campo com **AUTO\_INCREMENT**.
    - \* Na coluna/campo em que tem **AUTO\_INCREMENT**, insere-se o valor **NULL**, assim o **MySQL** entende que ele proprio deve auto incrementar aquele campo.

## 4.2 Consultando campos na tabela - SELECT

- O comando **SELECT** serve para projeção, seleção e junção.
- O comando **SELECT** seleciona os campos/colunas a serem mostrados.
- Projeta/constroi o que deve ser mostrado, não apenas os dados da tabela.
  - Exemplo de código:  
**SELECT 'SERGIO PEDRO' AS MEU\_NOME;**
  - Sintaxe:  
**SELECT 'algo a mostrar' AS alias\_da\_coluna;**
- Seleciona o que deve ser mostrado da tabela.
  - Exemplo de código:  
**SELECT NOME, SEXO, EMAIL, ENDERECO FROM CLIENTE;**
  - Sintaxe:  
**SELECT coluna\_1, coluna\_6, coluna\_3, coluna\_5 FROM tabela;**
  - Seleciona todas as colunas da tabela:  
**SELECT \* FROM tabela;**  
Obs.: '\*', Diminui a eficiência da pesquisa na tabela.

### 4.3 Consultando registros na tabela - WHERE

- O comando **WHERE** serve para filtrar os registros/linhas da tabela, antes de mostrar.
  - Sintaxe:  
**SELECT** *coluna\_1*, *coluna\_2* **FROM** *tabela*  
**WHERE** *coluna\_1* = *criterio*;
- O comando **WHERE** não precisa ter haver com a seleção **SELECT**.
  - Sintaxe:  
**SELECT** *coluna\_1*, *coluna\_3* **FROM** *tabela*  
**WHERE** *coluna\_2* = *criterio*;
- Para trabalhar com *strings*, é útil usar o comando **LIKE** e os *caracteres coringas*.
  - Caracteres coringas:
    - \* *'%'*  
Qualquer coisa.
    - \* *'\_'*  
Um único caracter.
  - Sintaxe:  
**SELECT** *coluna\_1*, *coluna\_3* **FROM** *tabela*  
**WHERE** *coluna\_2* **LIKE** *'string\_procurada'*;  
Obs.: Os caracteres coringas podem entrar em qualquer lugar da string para complementar o texto a procurar.
- Filtrando valores **NULL**.
  - Para filtrar valores **NULL**, basta utilizar o **IS NULL**, ao inves de *'= NULL'*.
    - \* Sintaxe:  
**SELECT** *coluna1*, *coluna2*, ... **FROM** *tabela*  
**WHERE** *colunaX* **IS NULL**;
  - Para filtrar valores não **NULL**, basta utilizar **IS NOT NULL**, ao inves de uma expressão.
    - \* Sintaxe:  
**SELECT** *coluna1*, *coluna2*, ... **FROM** *tabela*  
**WHERE** *colunaX* **IS NOT NULL**;

## 5 Módulo 5 - Operadores Lógicos, GROUP BY e ORDER BY

### 5.1 Operadores Lógicos e Performance de operadores lógicos

- Operadores lógicos:

- **OR**/OU

- \* Apenas uma condição precisa ser verdadeira para dar verdadeiro.

- \* Sintaxe:

- ```
SELECT * FROM tabela
WHERE (condição_1 OR condição_2);
```

- **AND**/E

- \* Todas as condições precisam ser verdadeiras para dar verdadeiro.

- \* Sintaxe:

- ```
SELECT * FROM tabela
WHERE (condição_1 AND condição_2);
```

- **NOT**/negação

- \* Nega e inverte e inverte o valor de uma expressão.

- \* Sintaxe:

- ```
SELECT * FROM tabela
WHERE (condição_1 AND NOT condição_2);
```

Obs.: Inverte o resultado da *condição\_2*.

- **IN**

- \* Lista determinados valores validos de uma coluna.

- \* Pode ser usado em conjunto com o operador **NOT**, para negar a lista (exceto a lista).

- \* Sintaxe:

- ```
UPDATE tabela SET coluna = valor_novo
WHERE coluna IN (valor_1, valor_2, ...);
```

- Tabela verdade

##	A	NOT_A	B	NOT_B	A_OR_B	A_AND_B
## 1	V	F	V	F	V	V
## 2	V	F	F	V	V	F
## 3	F	V	V	F	V	F
## 4	F	V	F	V	F	F

- Performance de operadores lógicos.

- Para melhorar a performance das consultas, com operadores lógicos, dois casos podem ser avaliados:

- \* No caso **OR**:

- Colocar a condição que oferece maior incidência de verdadeiro na frente.
- Se a primeira condição é verdadeira, a segunda não é avaliada, melhorando assim a performance da consulta.

\* No caso **AND**:

- Colocar a condição que oferece menor incidência de verdadeiro na frente.
- Se a primeira condição for falsa, a segunda nem é avaliada, pois o resultado é falso. Melhorando assim a performance da consulta.

## 5.2 Agregador e funções de agregação - GROUP BY

- **COUNT(\*)**

- Conta o numero de registros.
- Sintaxe:  
**SELECT COUNT (\*) FROM tabela;**

- **GROUP BY**

- Agrupa dados em torno de determinado campo.
- Usar em conjunto com funções de agrupamento, como:
  - \* **COUNT (\*)**  
Conta todos os registros.
  - \* **COUNT (coluna\_x)**  
Conta os registros da coluna x.
  - \* **AVG (coluna\_x)**  
Calcula a media dos valores da coluna x.
  - \* **MAX (coluna\_x)**  
Encontra o valor maximo da coluna x.
  - \* **MIN (coluna\_x)**  
Encontra o valor minimo da coluna x.
  - \* **SUM (coluna\_x)**  
Calcula a soma dos valores na coluna x.
- Sintaxe:  
**SELECT coluna\_x, COUNT(\*) FROM tabela  
GROUP BY coluna\_x;**
- É possível agrupar mais de uma coluna de uma vez.
  - \* A ordem em que as colunas aparecem na instrução **GROUP BY**, determinam a ordem de prioridade no agrupamento.
  - \* Sintaxe:  
**SELECT coluna1, coluna2,.. FROM tabela  
GROUP BY coluna1, coluna2;**  
Obs.: Prioridade primeiro agrupar a *coluna1*, depois agrupar em função da *coluna1* a *coluna2*.



## 5.3 Ordenando registros - ORDER BY

- ORDER BY

- Organiza os dados segundo uma ordem.
- Por default é ordem crescente, **ASC**.
- Para ordem decrescente só adicionar ao final **DESC**.
- Utilizado normalmente ao final de **WHERE** ou **GROUP BY**.
- Ao invés de colocar o nome da coluna, pode indicar a numeração da coluna na ordem em que aparece na instrução **SELECT**.
- Sintaxe:  
**SELECT** *coluna1, coluna2, ...* **FROM** *tabela*  
**GROUP BY** *coluna1*  
**ORDER BY** *coluna2*; (ou **ORDER BY** *2*;)
- Também é possível colocar em ordem, mais de uma coluna de uma vez.
  - \* A tabela é ordenada de acordo com a precedência em que as colunas aparecem no **ORDER BY**.
  - \* Sintaxe:  
**SELECT** *coluna1, coluna2, ...* **FROM** *tabela*  
**GROUP BY** *coluna1*  
**ORDER BY** *coluna2 ASC, coluna1 DESC*; (ou **ORDER BY** *2 ASC, 1 DESC*;)
- O comando **ORDER BY** também coloca em ordem **VIEWS**.

## 6 Módulo 7 - Mais comandos UPDATE e DELETE

### 6.1 Atualizando registros na tabela - UPDATE

- Atualizar todos os dados de uma coluna/campo de uma tabela, de uma vez.
  - Para atualizar todos os dados, de uma determinada coluna/campo, de uma tabela, para um dado determinado, basta usar **UPDATE** sem filtros.
  - Muito cuidado ao utilizar esse comando assim, pois pode gerar muitos problemas.
  - Sintaxe:  
**UPDATE** *tabela* **SET** *coluna\_a\_atualizar* = *valor\_atualizado*;
- Para atualizar um determinado registro.
  - Para atualizar um determinado dado de uma coluna/campo, utilizar o **UPDATE** em conjunto com a instrução **WHERE**.
  - Sintaxe:  
**UPDATE** *tabela* **SET** *coluna\_a\_atualizar* = *valor\_atualizado*  
**WHERE** *condição* = *valor*;

## 6.2 Deletando registros - DELETE

- Deletar todos os registros de uma tabela.
    - Sintaxe:  
**DELETE FROM** *tabela*;
  - Deletar apenas determinados registros de uma tabela, usar **DELETE** em conjunto com filtro **WHERE**.
    - Sintaxe:  
**DELETE FROM** *tabela*  
**WHERE** *critério\_do\_que\_se\_quer\_deletar* = *valor*;
  - Dicas:
    - Antes de deletar qualquer registro, deve-se conferir através de uma consulta, se os dados que aparecem são os que querem ser deletados.  
**SELECT \* FROM** *tabela*  
**WHERE** *mesmo\_critério\_do\_delete* = *valor*;
    - Contar os registros antes, durante a consulta e depois do **DELETE**. Para ter certeza sobre o que foi deletado.  
**SELECT COUNT(\*) FROM** *tabela*  
**WHERE** *mesmo\_critério\_do\_delete* = *valor*;
- Obs.: Exemplo de consulta de quantos registros devem ser deletados.

### 6.3 Transação - **START TRANSACTION**

- **START TRANSACTION;**

- As instruções dentro da transação, que serão avaliadas, ficam indentadas dentro da transação.
- Sintaxe:  
**START TRANSACTION;**  
*instrução\_1;*  
*instrução\_2;*  
...

- **COMMIT;**

- Aceita a transação (**START TRANSACTION;**). Confirma as instruções da transação.
- Fica fora da indentação da instrução **START TRANSACTION**.

- **ROLLBACK;**

- Nega a transação (**START TRANSACTION;**). Desfaz as instruções da transação.
- Instrução para voltar atrás em instruções.
- Desfaz instruções (como **UPDATE**, **DELETE**, ...), tudo que estiver dentro de **START TRANSACTION**.
- Fica fora da indentação da instrução **START TRANSACTION**.

Obs.: Essas instruções (**START TRANSACTION**, **COMMIT** e **ROLLBACK**) levam “;” ao final delas, não está errado como escrito a cima.

## 7 Módulo 8 - Modelagem

### 7.1 Primeira forma normal

- 3 Regras:
  1. Todo campo vetorizado se tornará outra tabela.
    - Campo vetorizado é todo campo que apresenta algo como um vetor dentro dele.
    - Varios dados do mesmo tipo (vetor).
    - Exemplo:  
*vetor* [VERDE, AMARELO, LARANJA,...]
  2. Todo campo multivalorado se tornará outra tabela.
    - Campo multivalorado é todo campo que apresenta algo como uma lista dentro dele.
    - Diversos dados de tipos diferentes (lista).
    - Exemplo:  
*list* (1, VERDE, CASA, ...)
  3. Toda tabela necessita de pelo menos um campo que identifique todo registro como sendo único (é o que chamamos de “**Chave Primaria**” ou “**Primary Key**”).
    - Tipos de **CHAVE PRIMARIA**:
      - \* NATURAL
        - Pertence ao registro intrinsecamente.
        - Muito útil, porem pouco confiavel. Depende de terceiros para existir, como o governo por exemplo.
        - Exemplo: CPF.
      - \* ARTIFICIAL
        - É criada pelo/para o banco de dados para identificar o registro.
        - Exemplo: ID.
        - Mais indicado de se trabalhar, pois oferece controle total por parte do administrador do banco de dados e não depende de terceiros para existir.

## 7.2 Segunda forma normal

“Uma relação está na **2º forma normal** se, e somente se, estiver na **1º forma normal** e cada atributo não-chave for dependente da chave primária inteira, isto é, cada atributo não-chave não poderá ser dependente de apenas parte da chave.”

- No caso de tabelas com chave primária composta, se um atributo depende apenas de uma parte da chave primária, então esse atributo deve ser colocado em outra tabela.
- Uma relação está na **2º forma normal** quando duas condições são satisfeitas:
  - A relação estiver na **1º forma normal**.
  - Todos os atributos primos dependerem funcionalmente de toda a **chave primária**.
- Conclusões:
  - Maior independência de dados.
  - Redundâncias e anomalias: dependências funcionais indiretas.

### 7.3 Terceira forma normal

“Uma relação R está na **3º forma normal** se ela estiver na **2º forma normal** e cada atributo não-chave de R não possuir **dependência transitiva**, para cada chave candidata de R. Todos os atributos dessa tabela devem ser independentes uns dos outros, ao mesmo tempo que devem ser dependentes exclusivamente da **chave primária** da tabela.”

- Exemplo ilustrativo:  
“Uma tabela não está na **Terceira Forma Normal** porque a coluna *Total* é dependente, ou é resultado, da multiplicação das colunas *Preço* e *Quantidade*, ou seja, a coluna *total* tem **dependência transitiva** de colunas que não fazem parte da **chave primária**, ou mesmo candidata da tabela. Para que essa tabela passe à **Terceira forma normal** o campo *Total* deverá ser eliminado, a fim de que nenhuma coluna tenha dependência de qualquer outra que não seja exclusivamente chave”.
- Passagem para a **3º forma normal**:
  - Para estar na **3º forma normal** precisa estar na **2º forma normal**.
  - Geração de novas tabelas com DF (Dependências Funcionais) diretas.
  - Análise de dependências funcionais entre atributos não-chave.
  - Verificar a dependência exclusiva da **chave primária**.
  - Entidades na **3º forma normal** também não podem conter atributos que sejam resultados de algum cálculo de outro atributo.
- Conclusões:
  - Maior independência de dados.
  - **3º forma normal** gera representações lógicas finais na maioria das vezes.
  - Redundâncias e anomalias: dependências funcionais.

## 8 Módulo 9 - PROJEÇÃO, SELEÇÃO E JUNÇÃO

Principais passos de uma consulta.

### 8.1 PROJEÇÃO

- O primeiro passo de uma consulta é montar o que quer ver na tela - **SELECT**.
- É tudo que você quer ver na tela.
- Sintaxe comentada:  
**SELECT** *coluna\_1* (PROJEÇÃO)  
**FROM** *tabela*; (ORIGEM)  
ou  
**SELECT** 2+2 **AS** *alias*; (PROJEÇÃO)  
Obs.: o que esta entre parênteses é comentario.

### 8.2 SELEÇÃO

- O segundo passo de uma consulta é a seleção dos dados de uma consulta - **WHERE**.
- É filtrar.
- Trazer um subconjunto do conjunto total de registros de uma tabela.
- Sintaxe comentada:  
**SELECT** *coluna\_1, coluna\_2, coluna\_3* (PROJEÇÃO)  
**FROM** *tabela* (ORIGEM)  
**WHERE** *critero = valor\_do\_criterio*; (SELEÇÃO)  
Obs.: o que esta entre parênteses é comentario.

### 8.3 JUNÇÃO

#### 8.3.1 Junção forma errada - gambiarra

- Usa seleção como uma forma de juntar tabelas.
- Como consequencia:
  - Uso de operadores lógicos para mais criterios de seleção - **WHERE**.
  - Ineficiencia na pesquisa, maior custo computacional.
- Sintaxe comentada:  
**SELECT** *coluna1\_tab1, coluna2\_tab1, coluna1\_tab2* (PROJEÇÃO)  
**FROM** *tabela1, tabela2* (ORIGENS)  
**WHERE** *chave\_primaria\_tab1 = chave\_estrangeira\_tab2*; (JUNÇÃO)  
ou  
**SELECT** *coluna1\_tab1, coluna2\_tab1, coluna1\_tab2* (PROJEÇÃO)  
**FROM** *tabela1, tabela2* (ORIGENS)  
**WHERE** *chave\_primaria\_tab1 = chave\_estrangeira\_tab2* (JUNÇÃO)  
**AND** *critério = valor*; (SELEÇÃO com operador lógico)



Obs.: o que esta entre parênteses é comentario.

### 8.3.2 Junção forma certa - JOIN

- Junção **JOIN**, junta duas ou mais tabelas apartir das colunas de *chaves primarias* e *chaves estrangeiras*.
- Admite seleção - **WHERE** - sem maiores custos computacionais.

#### 8.3.2.1 INNER

- Exclui os registros sem par (orfans) na outra tabela - **INNER**.
- Consulta com duas tabelas.

– Sintaxe comentada:

```
SELECT coluna1_tab1, coluna2_tab1, coluna1_tab2 (PROJEÇÃO)
FROM tabela1 (ORIGEM)
INNER JOIN tabela2 (JUNÇÃO)
ON chave_primaria_tab1 = chave_estrangeira_tab2
WHERE criterio = valor;(SELEÇÃO)
```

#### 8.3.2.2 LEFT

- Mostra ate os registros sem par (nulos) - **LEFT**.
  - Comum usar a função *IFNULL()* para tratar os valores nulos.
- Consulta com duas tabelas.

– Sintaxe comentada:

```
SELECT coluna1_tab1, coluna2_tab1, coluna1_tab2 (PROJEÇÃO)
FROM tabela1 (ORIGEM)
LEFT JOIN tabela2 (JUNÇÃO)
ON chave_primaria_tab1 = chave_estrangeira_tab2
WHERE criterio = valor;(SELEÇÃO)
```

### 8.3.2.3 Cláusulas ambíguas e Ponteiramento

- Consulta com mais de duas tabelas.
  - Pode apresentar colunas/campos com o mesmo nome, de tabelas diferentes. Caso comum das *chaves estrangeiras* (**FK**).
  - Indicar de onde vem cada coluna através de “*nome\_da\_tabela.nome\_da\_coluna*”.
  - Sintaxe comentada:  
**SELECT**  
*tabela1.coluna1\_tab1,*  
*tabela1.coluna2\_tab1,*  
*tabela2.coluna1\_tab2,*  
*tabela3.coluna1\_tab3* (PROJEÇÃO)  
**FROM** *tabela1* (ORIGEM)  
**LEFT JOIN** *tabela2* (JUNÇÃO)  
**ON** *tabela1.chave\_primaria\_tab1 = tabela2.chave\_estrangeira\_tab2*  
**INNER JOIN** *tabela3* (JUNÇÃO)  
**ON** *tabela1.chave\_primaria\_tab1 = tabela3.chave\_estrangeira\_tab3*  
**WHERE** *criterio = valor*;(SELEÇÃO)  
Obs.: o que esta entre parênteses é comentario.
- Ponteiramento (alias para tabelas)
  - Melhora a performance da consulta.
  - Sintaxe comentada:  
**SELECT**  
*A.coluna1\_tab1,*  
*A.coluna2\_tab1,*  
*B.coluna1\_tab2,*  
*C.coluna1\_tab3*  
**FROM** *tabela1 A* (PONTEIRAMENTO DA TABELA 1)  
**LEFT JOIN** *tabela2 B* (PONTEIRAMENTO DA TABELA 2)  
**ON** *A.chave\_primaria\_tab1 = B.chave\_estrangeira\_tab2*  
**INNER JOIN** *tabela3 C* (PONTEIRAMENTO DA TABELA 3)  
**ON** *A.chave\_primaria\_tab1 = C.chave\_estrangeira\_tab3*  
**WHERE** *criterio = valor*;

## 9 Categoria de comandos

### 9.1 DML - *Data Manipulation Language* (Linguagem de Manipulação de Dados)

É um conjunto de instruções usada nas consultas e modificações dos dados armazenados nas tabelas do banco de dados.

- **INSERT**

- Adiciona registros numa tabela.

- Sintaxe:

- INSERT INTO** *nome\_da\_tabela*

- VALUES**

- (valor\_na\_coluna\_1\_registro1, valor\_na\_coluna\_2\_registro1,...)*,

- (valor\_na\_coluna\_1\_registro2, valor\_na\_coluna\_2\_registro2,...)*,

- ...;*

- **UPDATE**

- Altera os dados de um ou mais registros em uma tabela.

- Sintaxe:

- UPDATE** *tabela* **SET** *coluna\_a\_atualizar = valor\_atualizado*

- WHERE** *condição = valor;*

- **DELETE**

- Remove um ou mais registros de uma tabela.

- Sintaxe:

- DELETE FROM** *tabela*

- WHERE** *critério\_do\_que\_se\_quer\_deletar = valor;*

## 9.2 DDL - *Data Definition Language* (Linguagem de definição de dados)

É um conjunto de instruções usado para criar e modificar as estruturas dos objetos armazenados no banco de dados.

- **CREATE**

Utilizada para construir um novo banco de dados, tabela, índice ou consulta armazenada.

- **DATABASE**

- \* Criação de banco de dados.

- \* Sintaxe:

- CREATE DATABASE** *nome\_banco\_de\_dados*;

- **TABLE**

- \* Criação de tabela.

- \* Sintaxe:

- CREATE TABLE** *nome\_tabela* (  
*coluna1* *tipo regra restrições*,  
*coluna2* *tipo regra restrições*,  
...  
);

- **DROP**

Remove um banco de dados, tabela, índice ou visão existente.

- **DATABASE**

- \* Remove banco de dados.

- \* Sintaxe:

- DROP DATABASE** *nome\_do\_banco\_de\_dados*;

- **TABLE**

- \* Remove tabela.

- \* Sintaxe:

- DROP TABLE** *nome\_da\_tabela*;

- **ALTER**

- Modifica um objeto existente do banco de dados.

- É possível incluir, eliminar e alterar colunas.

- Para alterar uma tabela existente, é necessário que os registros existentes já sejam compatíveis com a alteração.

\* **CHANGE**

- Altera o nome e o tipo da coluna/campo.
- Para alterar apenas o tipo, é necessário repetir o nome da coluna/campo.
- Sintaxe:  
**ALTER TABLE** *nome\_tabela*  
**CHANGE** *nome\_coluna* (*novo*)*nome\_coluna modificação\_tipo*;

\* **MODIFY**

- Altera o tipo e regras de uma coluna/campo.
- Sintaxe:  
**ALTER TABLE** *nome\_tabela*  
**MODIFY** *nome\_coluna modificação\_tipo*;

\* **ADD**

- Adiciona chaves (primária ou estrangeira) a uma coluna.
- Não é possível adicionar “*auto\_increment*”.
- Sintaxe:  
**ALTER TABLE** *tabela*  
**ADD PRIMARY KEY**(*coluna*);  
ou  
**ALTER TABLE** *tabela*  
**ADD FOREIGN KEY**(*coluna\_da\_tabela*)  
**REFERENCES** (*coluna\_chave\_primaria\_de\_outra\_tabela*);
- O comando **ADD** funciona como abreviado do comando **ADD COLUMN**.
- Sintaxe:  
**ALTER TABLE** *tabela*  
**ADD** *nova\_coluna tipo*;

\* **ADD COLUMN**

- Adicionando uma nova coluna.
- Sintaxe:  
**ALTER TABLE** [*nome\_database.*]*nome\_tabela*  
**ADD COLUMN** *nome\_coluna tipo*;
- Para alterar a posição de entrada da coluna na tabela, usar **FIRST** (para aparecer na primeira posição da tabela) ou **AFTER** (depois de tal coluna).
- Sintaxe:  
**ALTER TABLE** [*nome\_database.*]*nome\_tabela*  
**ADD COLUMN** *nome\_coluna tipo*  
**FIRST**;

ou  
**ALTER TABLE** [nome\_database.]*nome\_tabela*  
**ADD COLUMN** *nome\_coluna tipo*  
**AFTER** *coluna\_de\_referencia*;

**\* DROP COLUMN**

- Deleta uma determinada coluna de uma tabela.
- Sintaxe:  
**ALTER TABLE** [nome\_database.]*nome\_tabela*  
**DROP COLUMN** *nome\_coluna*;

**\* RENAME**

- Renomeia o nome de uma tabela.
- Sintaxe:  
**ALTER TABLE** *tabela*  
**RENAME** *novo\_nome\_tabela*;

**• TRUNCATE**

- Esvazia imediatamente todo o conteúdo de uma tabela ou objeto que contenha dados.
- É muito mais rápido que um comando DELETE, pois, ao contrário deste, não armazena os dados sendo removidos no log de transações. Por esse motivo, em vários SGBDs é um comando não-transacional e irrecuperável, não sendo possível desfazê-lo com **ROLLBACK**.
- Sintaxe:  
**TRUNCATE TABLE** *nome\_tabela*;

**• RENAME**

- Mudar nome da tabela e/ou database.
- Sintaxe:  
**RENAME TABLE** *nome\_database.nome\_tabela* **TO** *nome\_database.novo\_nome\_tabela*;  
ou  
**RENAME TABLE** *nome\_database.nome\_tabela* **TO** *novo\_nome\_database.nome\_tabela*;

### 9.3 DCL - *Data Control Language* (Linguagem de Controle de Dados)

São usados para controle de acesso e gerenciamento de permissões para usuários em no banco de dados. Com eles, pode facilmente permitir ou negar algumas ações para usuários nas tabelas ou registros (segurança de nível de linha).

- USER - usuário

- **CREATE USER**

- \* Comando para criação de usuários.

- \* Determina user = usuário, host = local (IP do servidor ou *localhost* - maquina local) e password = senha.

- \* Sintaxe:

- CREATE USER** '*user*'@'*host*' **IDENTIFIED BY** '*password*';

- Listar usuários:

- SELECT user FROM mysql.user;**

- Mostrar usuário conectado atual:

- SELECT user();**

- Removendo usuários:

- DROP USER** '*exemplo*'@'*host*';

- Conectando ao MySQL por um usuário:

- mysql -u nome\_usuário -p password**

- GRANT

- Permitir que usuários especificados realizem tarefas especificadas.

- Também permite gerenciar permissão para realizar tarefas específicas em database e/ou tabelas específicas.

- Sintaxe:

- GRANT** *tipo\_de\_permissão* **ON** *nome\_database.nome\_tabela* **TO** '*username*'@'*localhost*';

- ou para dar permissão de root:

- GRANT ALL PRIVILEGES ON \* . \* TO** '*newuser*'@'*localhost*';

- Carregar/atualizar permissões:

- FLUSH PRIVILEGES;**

- Revisar as permissões atuais de um usuário:

- SHOW GRANTS FOR** '*username*'@'*localhost*';

- REVOKE

- Cancela/revoga permissões previamente concedidas.

- Sintaxe:  
**REVOKE** *tipo\_de\_permissão* **ON** *nome\_database.nome\_tabela* **FROM** 'username'@'localhost';  
Obs.: Note que no **REVOKE** é usado **FROM** e no **GRANT** é usado **TO**.

- Privilégios que podem ser CONCEDIDOS à ou REVOCADOS de um usuário:
  - **ALL PRIVILEGES** — como vimos anteriormente, isso garante ao usuário do MySQL acesso completo a um banco de dados (ou, se nenhum banco de dados for selecionado, acesso global a todo o sistema).
  - **CREATE** — permite criar novas tabelas ou bancos de dados.
  - **DROP** — permite deletar tabelas ou bancos de dados.
  - **DELETE** — permite excluir linhas de tabelas.
  - **INSERT** — permite inserir linhas em tabelas.
  - **SELECT** - permite usar o comando SELECT para ler os bancos de dados.
  - **UPDATE** — permite atualizar linhas de tabelas.
  - **GRANT OPTION** — permite conceder ou remover privilégios de outros usuários.

Outras instruções:

- **CONNECT**
- **EXECUTE**
- **USAGE**



## 9.4 TCL - *Tool Command Language* (Linguagem de Comandos de Ferramentas)

São usados para gerenciar as mudanças feitas por instruções DML. Ele permite que as declarações a serem agrupadas em transações lógicas.

- **START TRANSACTION**

- O comando garante que diversas instruções sejam executadas, porém se alguma for mal sucedida todas falham.
- É possível avaliar o processo de implementação das instruções e seus resultados e caso necessário regressar ao estado anterior as instruções ou confirmar sua implementação.
- Principais instruções que são comuns de serem usadas na transação são as **DML (INSERT, UPDATE e DELETE)**.
- Sintaxe:  
**START TRANSACTION;**

- **BACKROLL**

- Regressão para o estado anterior ao início da transação (**START TRANSACTION**).
- Sintaxe:  
**BACKROLL;**

- **COMMIT**

- Confirmação de que as instruções da transação (**START TRANSACTION**) podem ser implementadas sem problemas.
- Sintaxe:  
**COMMIT;**

## 10 Módulo 11 - Funções e VIEWS

### 10.1 Funções

Função é um bloco de programação que executa algo.

- **IFNULL()**

- Converte os valores **NULL** de uma coluna em um valor-padrão especificado.
- Os argumentos da função são a coluna a ser checada e o valor-padrão.
- Se o valor-padrão for um texto, ele entra entre aspas (*'valor-padrão'*).
- Uma observação é quanto ao cabeçalho da coluna/campo, o ideal é que ele seja modificado com uso do **AS** para um novo nome, senão ele imprime em tela a formulação que esta passando a coluna.
- É igual a função *coalesce()* em SQL.
- Sintaxe:  
**SELECT**  
...  
**IFNULL**(*coluna,valor-padrão*) **AS** *novo\_nome\_coluna*,  
...

- Funções de tempo

- **NOW()**
  - \* Função que retorna data e hora do sistema do computador.
  - \* Formato 'AAAA-MM-DD HH:MM:SS'.
  - \* É possível fazer operações com a data usando operador desejado, '**INTERVAL**' e adicionando o que deseja trabalhar (ex.: 1 DAY).
  - \* A função **NOW()** Pode ser usada como argumento das outras funções de tempo, para pegar o momento atual do sistema.
  - \* Sintaxe:  
**SELECT NOW()** **AS** *alias*,  
**NOW()** + **INTERVAL** 1 **DAY** **AS** *alias*;
- **TIME()**  
Retorna apenas a parte em formato de tempo 'HH:MM:SS'.
- **DATE()**  
Retorna apenas a parte em formato de data 'AAAA-MM-DD'.
- **YEAR()**  
Retorna apenas a parte em formato de anos.
- **MONTH()**  
Retorna apenas a parte em formato de meses.

- **DAY()**  
Retorna apenas a parte em formato de dias.
- **HOURL()**  
Retorna apenas a parte em formato de horas.
- **MINUTE()**  
Retorna apenas a parte em formato de minutos.
- **SECOND()**  
Retorna apenas a parte em formato de segundos.

Obs.: **INTERVAL** é usado para operações em todas essas funções de tempo.

- Função para descobrir usuario
  - **CURRENT\_USER()**  
Retorna o nome de usuário e o nome do host da conta MySQL que é usada pelo servidor para autenticar o cliente atual. Em resumo o cliente atual.

## 10.2 VIEWS

### 10.2.1 DDL VIEW

- Quando salvamos uma consulta em um banco de dados, ela se chama **VIEW**.
- Uma **VIEW** se comporta de forma semelhante a uma tabela, para todos os efeitos.
- Perde um pouco de performance da consulta, porem ganha em desenvolvimento da consulta.
- Criando **VIEW**
  - As VIEWS ficam salvas junto das tabelas, logo para consulta-las é necessario usar o ‘**SHOW TABLES;**’.
  - Por conta de onde fica armazenada as VIEWS se torna necessario dar um nome diferente para criar um diferenciação, normalmente é usado o prefixo ‘*VW\_*’, ex.: *VW\_nome\_da\_view*.
  - Sintaxe:  
**CREATE VIEW** *VW\_nome\_da\_view* **AS**  
**SELECT**  
...  
**FROM** *nome\_tabela*  
...;
- Apagando uma **VIEW**
  - Sintaxe:  
**DROP VIEW** *VW\_nome\_da\_view*;

### 10.2.2 DML VIEW

- Consultando uma **VIEW** - **SELECT** e **WHERE**
  - Como a **VIEW** funciona como uma tabela do banco de dados, é possível fazer consulta na **VIEW**, ao invés de consultar alguma tabela do banco de dados.
  - Funciona de maneira semelhante a consulta numa tabela.
  - Sintaxe:  
**SELECT**  
...  
**FROM** *VW\_nome\_da\_view*  
...  
**WHERE** *coluna = criterio*;
- Não dá para fazer **INSERT** e **DELETE** em **VIEW** formada por **JOIN**, que junta duas ou mais tabelas.
- Porém **UPDATE** é possível fazer.
- **VIEWS** sem **JOIN**, não tem restrição quanto ao **INSERT** e **DELETE**.
- Alterar a **VIEW** altera as tabelas que ela aponta. CUIDADO!

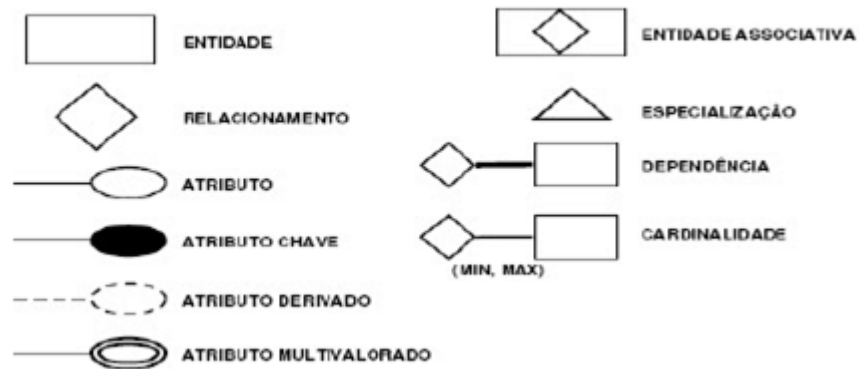
## 11 Módulo 12 - Diagrama ER - brModelo e StarUML

- Existem dois tipo de notação para diagrama ER (Entidade Relacionamento):
  - Peter Chen
    - \* Esse mais utilizado em literatura sobre banco de dados.
    - \* Software:  
**brModelo**
  - Cross foot
    - \* Vantagem do diagrama ser menos poluido.
    - \* Esse mais utilizado por arquitetos de dados.
    - \* Software:  
**StarUML**

## 11.1 Peter Chen

- Notação do Peter Chen

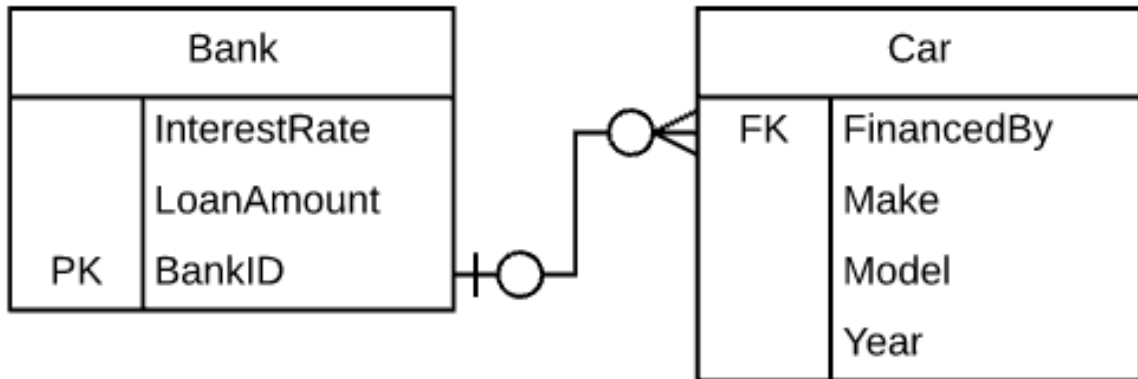
### Notação Peter Chen



- Entidade = Tabela
- Relacionamento = Relacionamento entre tabelas
- Atributo = Coluna/Campo
- Cardinalidade (x,y):
  - \* x = Obrigatoriedade (“0” não obrigatorio, “1” obrigatorio)
  - \* y = Tipo de relacionamento (“N” para muitos, “1” para um)

## 11.2 Cross Foot (pé de galinha)

- Entidades



- PK = Primary Key (Chave Primária)
- FK = Foreign Key (Chave Estrangeira)

- Atributos e Tipos

Entity
Field
Field
Field

Entity	
Key	Field
Key	Field
Key	Field

Entity	
Field	Type
Field	Type
Field	Type

Entity		
Key	Field	Type
Key	Field	Type
Key	Field	Type

- Cardinalidade



One



Many



One (and only one)



Zero or one



One or many



Zero or many

Obs.: Para inserir cardinalidade, deve clicar e arrastar o mouse entre as entidades.



## 12 Módulo 13 - DELIMITER e STORED PROCEDURES

### 12.1 Como mudar o delimitador

- O delimitador serve para indicar ao banco de dados o final de uma instrução.
- Por padrão o delimitador do **MySQL** é o “;” (ponto e virgula).
- Dá para verificar o delimitador em uso através do comando **STATUS**.
- Porém é possível mudar o delimitador para poder programar no **MySQL**.
  - O delimitador é apenas um caractere.
  - É um comando de infraestrutura, logo não precisar de delimitador no final.
  - Sintaxe:  
**DELIMITER** *novo\_caractere*

## 12.2 STORED PROCEDURES - Procedimentos Armazenados - Funções

### 12.2.1 Bloco anônimo

- Blocos anônimos não são armazenados.
- São instruções simples que servem apenas para serem executadas uma única vez, como uma consulta pontual e etc.

### 12.2.2 Blocos nomeados

- Blocos nomeados são **STORED PROCEDURES**, procedimentos armazenadas (funções programadas com instruções, armazenadas pelo sistema).
- São blocos de programação (instruções) que serão usados varias vezes.
- Criando função (**CREATE PROCEDURE**)
  - É necessario mudar o delimitador para não confundir o delimitador do final da função com das instruções.
  - Sintaxe:  
**DELIMITER \$**  
**CREATE PROCEDURE** *nome\_função()*  
**BEGIN**  
instruções;  
...  
**END**  
**\$**  
Obs.: As instruções internas da função estão com o delimitador padrão “;”, enquanto que a **CREATE PROCEDURE** termina com o novo delimitador “\$”, para diferenciar o que é um e o que é o outro para o sistema.
- Chamando uma função (Chamando uma **PROCEDURE**)
  - Posso voltar com meu delimitador para o padrão “;”.
  - Sintaxe:  
**DELIMITER ;**  
**CALL** *nome\_função()*;
- Criando uma função que recebe parametros.
  - É necessario determinar qual o *tipo* de dado de cada *parametro* (ver Módulo 2).
  - Sintaxe:  
**DELIMITER \$**  
**CREATE PROCEDURE** *nome\_função(parametro1 tipo, parametro2 tipo)*  
**BEGIN**  
instruções com os parametros;  
...

**END**  
\$

- Chamando uma função com parametros (Chamando uma **PROCEDURE**)

- Posso voltar com meu delimitador para o padrão “;”.

- Sintaxe:

- DELIMITER ;**

- CALL** *nome\_função*(*parametro1*, ...);

- Apagar uma função.

- Sintaxe:

- DROP PROCEDURE** *nome\_função*;

- Obs.: Sem os “()” da função.

### 12.2.3 Problemas de usar PROCEDURES

- Cada banco de dados (**MySQL**, **ORACLE**,... ) tem sua linguagem de programação, logo dificulta a migração de banco de dados.
- As regras de negócio ficam atreladas ao banco de dados, não é uma boa pratica.

### 12.2.4 Pontos positivos de usar PROCEDURES

- Desafoga a área de controle (**C#**, **JAVA**, **JS**, **Ruby**, **PHP**,...) do sistema a qual se esta trabalhando.
- Pode ser uma boa saída para melhorar o desempenho da área de controle (linguagens de programação), distribuir as regras de negócio entre controle e banco de dados.

## 13 Módulo 14 - Funções Básicas

- **COUNT** (\*)
  - Conta todos os registros.
- **COUNT** (*coluna\_x*)
  - Conta os registros da coluna x.
- **AVG** (*coluna\_x*)
  - Calcula a media dos valores da coluna x.
- **MAX** (*coluna\_x*)
  - Encontra o valor maximo da coluna x.
- **MIN** (*coluna\_x*)
  - Encontra o valor minimo da coluna x.
- **SUM** (*coluna\_x*)
  - Calcula a soma dos valores na coluna x.
- **TRUNCATE** (*numero, numero\_casa\_decimais*)
  - Trunca o numero para um numero com as casas decimais estabelecidos.
  - O numero pode ser uma função que calculou algo a partir de uma coluna (**AVG**, **SUM**, ...).
  - Não confundir com a função **TRUNCATE TABLE**.

## 14 Módulo 15 - Subqueries (Subconsulta) e Trabalhando com linhas

### 14.1 Subqueries (Subconsulta)

- Uma consulta dentro do resultado de outra consulta.
- Pode ser usado como o filtro de uma nova consulta, quando usado dentro do **WHERE**.
  - O retorno de colunas da segunda consulta deve ser igual ao numero de colunas do filtro.
  - Sintaxe:  
**SELECT**  
*coluna1*  
...  
**FROM** *tabela*  
**WHERE** *coluna1* = (**SELECT** *coluna* **FROM** *tabela* **WHERE** *coluna\_x* = *critério*);

### 14.2 Trabalhando com linhas

- Não tem funções específicas para trabalhar com linhas/registros.
- Porém através da projeção (**SELECT**) é possível manipular novas colunas.
- Sintaxe:  
**SELECT**  
*coluna\_1*,  
...  
**TRUNCATE**(*coluna\_1*+*coluna\_2*+.../10, 2) **AS** “Media”  
**FROM** *tabela*;

## 15 Módulo 16 - Modificação de tabelas

### 15.1 Modificação de tabelas - ALTER

- **ALTER**

- Modifica um objeto existente do banco de dados.
- É possível incluir, eliminar e alterar colunas.
- Para alterar uma tabela existente, é necessário que os registros existentes já sejam compatíveis com a alteração.

- \* **CHANGE**

- Altera o nome e o tipo da coluna/campo.
- Para alterar apenas o tipo, é necessário repetir o nome da coluna/campo.
- Sintaxe:  
**ALTER TABLE** *nome\_tabela*  
**CHANGE** *nome\_coluna* (*novo*)*nome\_coluna* *modificação\_tipo*;

- \* **MODIFY**

- Altera o tipo e regras de uma coluna/campo.
- Sintaxe:  
**ALTER TABLE** *nome\_tabela*  
**MODIFY** *nome\_coluna* *modificação\_tipo*;

- \* **ADD**

- Adiciona chaves (primária ou estrangeira) a uma coluna.
- Não é possível adicionar “*auto\_increment*”.
- Sintaxe:  
**ALTER TABLE** *tabela*  
**ADD PRIMARY KEY**(*coluna*);  
ou  
**ALTER TABLE** *tabela*  
**ADD FOREIGN KEY**(*coluna\_da\_tabela*)  
**REFERENCES** (*coluna\_chave\_primaria\_de\_outra\_tabela*);
- O comando **ADD** funciona como abreviado do comando **ADD COLUMN**.
- Sintaxe:  
**ALTER TABLE** *tabela*  
**ADD** *nova\_coluna* *tipo*;

- \* **ADD COLUMN**

- Adicionando uma nova coluna.

- Sintaxe:  
**ALTER TABLE** [nome\_database.]*nome\_tabela*  
**ADD COLUMN** *nome\_coluna tipo*;
- Para alterar a posição de entrada da coluna na tabela, usar **FIRST** (para aparecer na primeira posição da tabela) ou **AFTER** (depois de tal coluna).
- Sintaxe:  
**ALTER TABLE** [nome\_database.]*nome\_tabela*  
**ADD COLUMN** *nome\_coluna tipo*  
**FIRST**;  
ou  
**ALTER TABLE** [nome\_database.]*nome\_tabela*  
**ADD COLUMN** *nome\_coluna tipo*  
**AFTER** *coluna\_de\_referencia*;
- \* **DROP COLUMN**
  - Deleta uma determinada coluna de uma tabela.
  - Sintaxe:  
**ALTER TABLE** [nome\_database.]*nome\_tabela*  
**DROP COLUMN** *nome\_coluna*;
- \* **RENAME**
  - Renomeia o nome de uma tabela.
  - Sintaxe:  
**ALTER TABLE** *tabela*  
**RENAME** *novo\_nome\_tabela*;
- \* **CONSTRAINTS**
  - Cria regras, muito usado para adicionar chaves (PK e FK) a tabela.
  - Sintaxe:  
**ALTER TABLE** *nome\_tabela*  
**ADD CONSTRAINTS** *nome\_da\_regra*  
**PRIMARY KEY**(*coluna\_chave\_primaria*);  
ou  
**ALTER TABLE** *nome\_tabela*  
**ADD CONSTRAINTS** *nome\_da\_regra*  
**FOREIGN KEY**(*coluna\_chave\_estrangeira*) **REFERENCES** *tabela\_chave\_primaria*(*coluna\_chave\_p*);
- \* **DROP CONSTRAINTS**
  - Apaga regras.
  - Sintaxe:  
**ALTER TABLE** *nome\_tabela*  
**DROP FOREIGN KEY** *nome\_da\_regra*;  
ou

```
ALTER TABLE nome_tabela  
DROP PRIMARY KEY nome_da_regra;
```

- **RENAME**

- Mudar nome da tabela e/ou database.

- Sintaxe:

- RENAME TABLE** *nome\_database.nome\_tabela* **TO** *nome\_database.novo\_nome\_tabela*;

- ou

- RENAME TABLE** *nome\_database.nome\_tabela* **TO** *novo\_nome\_database.nome\_tabela*;



## 15.2 Constraints - regras e boas praticas

- Para poder visualizar de maneira mais organizada atraves do *dicionario de dados*, é interessante adicionar as chaves fora da criação de tabelas.
- Ao adicionar a chave dentro da criação de tabelas o sistema dá um nome automatico para a chave no sistema. O que não é desejado e pode ficar confuso.
- Ao adicionar a chave fora da criação de tabelas o usuario determina o nome daquela chave que ficara gravada no sistema.
- Os nomes das chaves podem ser consultados no *dicionario de dados* do sistema e no:  
**SHOW CREATE TABLE** *nome\_da\_tabela*;
- Boas praticas:
  - Criar primeiro as tabelas, **CREATE TABLE**.
  - Depois criar as chaves primarias e estrangeiras.
  - Nome da regra, serve para nomear esta regra no dicionario de dados.
  - Uma boa pratica é nomear a regra em *chave primaria*(PK) como **PK\_\_**(*tabela\_da\_PK*), sem os paranteses.
  - Uma boa pratica é nomear a regra em *chave estrangeira*(FK) como **FK\_\_**(*tabela\_da\_PK*)\_\_(*tabela\_da\_FK*), sem os paranteses.
  - Sintaxe:  
**ALTER TABLE** *nome\_tabela*  
**ADD CONSTRAINTS** *nome\_da\_regra*  
**PRIMARY KEY**(*coluna\_chave\_primaria*);  
ou  
**ALTER TABLE** *nome\_tabela*  
**ADD CONSTRAINTS** *nome\_da\_regra*  
**FOREIGN KEY**(*coluna\_chave\_estrangeira*) **REFERENCES** *tabela\_chave\_primaria*(*coluna\_chave\_primaria*)

### 15.3 Dicionario de dados do sistema

- O dicionario de dados é o **metadado**, os dados sobre os dados (como nome das tabelas, data de criação, responsavel pela criação,...).
- O dicionario de dados é constituído no **MySQL** pelas **DATABASES** (**SHOW DATABASES**):

- *information\_schema*

- \* **CONSTRAINTS** (TABLES\_CONSTRAINTS)

- \* **TRIGGERS** (TRIGGERS)

- *mysql*

- *performance\_schema*

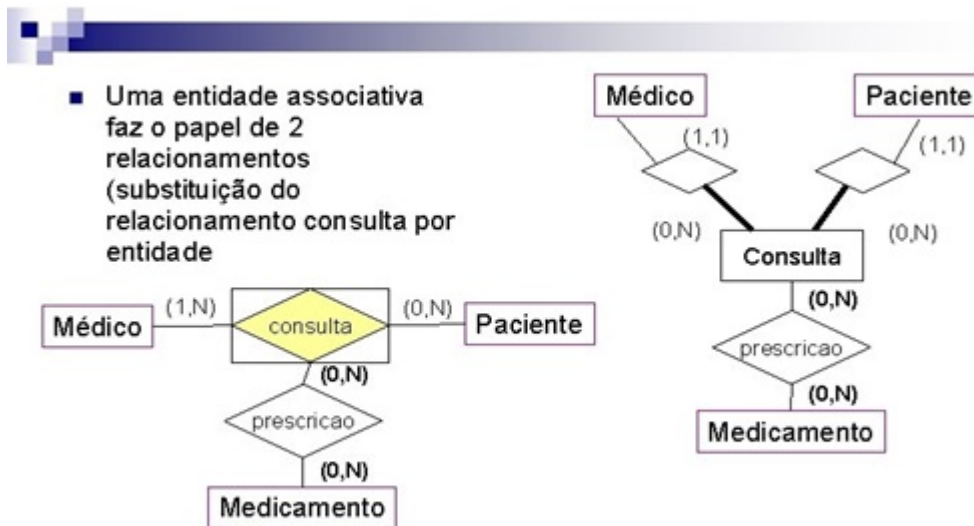
- Para averiguar as tabelas dentro da **DATABASE** *dicionario de dados* basta usar **DESC** (descrição da tabela) e **SELECT** (verificar os dados/registros contidos na tabela, basta fazer uma consulta normal na tabela do dicionario de dados).

## 16 Módulo 18 - Entidade Associativa e Chaves

### 16.1 Entidades Associativas

- Entidades associativas aparecem quando temos uma relação entre entidades do tipo N:N (muitos para muitos).
- Na entidade associativa, o relacionamento N:N (muitos para muitos) foi dividido em dois relacionamentos do tipo 1:N (um para muitos), sendo que a entidade associativa passa a servir de intermediário entre as entidades.

### Entidade Associativa



- Esta entidade é composta pelas chaves das duas entidades principais.
- Se fosse necessário, nesta entidade (associativa) também poderíamos adicionar informações complementares como quantidade, e outros campos.

## 16.2 Sobre Chaves

- *Chave Primaria (PK)*
  - No caso da entidade associativa, podemos definir que os campos principais da tabela funcionam como uma *chaves primarias (PK)*.
  - São definidas assim porque é comum que o resultado da combinação dos campos não possam se repetir, formando assim uma identidade unica, criada a partir da combinação de campos.
  - Sintaxe:  
**ALTER TABLE** *tabela\_associativa*  
**ADD CONSTRAINTS** *PK\_tabela\_associativa*  
**PRIMARY KEY** (*campo1,campo2,...*);
- *Chave Estrangeira (FK)*
  - Alem de *chaves primarias (PK)*, os campos principais da entidade associativa, também referenciam a chaves primarias das entidades/tabelas que ela quer juntar, logo também são *chaves estrangeiras (FK)*.
  - Não tem problema, e nem é incomum, uma *chave primaria (PK)* ser também um *chave estrangeira (FK)* nesses casos.

## 17 Módulo 19 - TRIGGERS (Gatilhos)

### 17.1 TRIGGERS

- A **TRIGGER** é um gatilho de programação, que dispara toda vez que algo predeterminado acontecer.
- Exemplos de gatilhos disparadores de uma **TRIGGER** são:
  - **INSERT**
  - **UPDATE**
  - **DELETE**
- Apos os gatilhos (**TRIGGERS**) disparados, são executados blocos de programação.
- Sintaxe:  
**DELIMITER \$**  
**CREATE TRIGGER** *nome\_da\_trigger*  
**BEFORE/AFTER INSERT/DELETE/UPDATE ON** *tabela*  
**FOR EACH ROW** (para cada linha)  
**BEGIN**  
(bloco de programação, qualquer comando SQL)  
**END**  
**DELIMITER ;**
- Ao inserir um comando SQL no bloco de programação para ser executada, é preciso terminar cada instrução com o delimitador “;”, logo é preciso mudar o delimitador para programar o **TRIGGER**.
- Problema do **BEFORE/INSERT**:
  - Quando o usado o **BEFORE** (antes) em conjunto com o **INSERT**, o **TRIGGER** pega o dado antes de ir para a tabela, logo o campo/coluna com **AUTO\_INCREMENT**, não gerou o numero ainda na tabela, então o **TRIGGER** pega o valor 0, nesse tipo de campo.
  - Para pegar o valor com **AUTO\_INCREMENT** no **INSERT**, pelo **TRIGGER**, basta usar o **AFTER** (depois) para pegar o novo valor. Pois os dados só são pegos pelo **TRIGGER** depois de os dados do **INSERT** terem entrado na tabela, e o novo valor no campo com **AUTO\_INCREMENT** ter sido gerado.

### 17.2 Deletando o TRIGGER

- Deletando um **TRIGGER**:  
**DROP TRIGGER** *nome\_do\_trigger*;

### 17.3 Conceito de NEW e OLD

- Definição:

- **OLD.coluna**  
Pega o valor antigo da coluna indicada.
  - **NEW.coluna**  
Pega o novo valor da coluna indicada.
- Usado dentro da instrução de comando **SQL**, no bloco de programação, na criação do **TRIGGER**.
  - Sintaxe:  
**DELIMITER \$**  
**CREATE TRIGGER** *nome\_da\_trigger*  
**BEFORE/AFTER INSERT/DELETE/UPDATE ON** *tabela\_observada\_pelo\_trigger*  
**FOR EACH ROW** (para cada linha)  
**BEGIN**  
**INSERT INTO** *tabela\_de\_ação\_do\_trigger*  
**VALUES**  
(NULL, **OLD.coluna1**, **OLD.coluna2**, **OLD.coluna3**);  
**END**  
**DELIMITER ;**

## 17.4 Observações TRIGGER

- A “*tabela\_observada\_pelo\_trigger*” é a tabela que vai dar gatilho ao TRIGGER.
- A “*tabela\_de\_ação\_do\_trigger*” é a tabela que vai sofrer alguma ação especificada pelo SQL, do bloco de programação.

## 17.5 Uso de **TRIGGER** para **BACKUP**

- Uma das utilidades mais apreciadas do uso de **TRIGGERS** é para fazer backup de ações.
- É uma boa pratica criar um banco de dados (**DATABASE**) só para backup de tabelas.
  - Lembrar que para comunicar um **TRIGGER** entre bancos de dados (**DATABASE**) é preciso mudar a forma de escrever o nome da tabela (ver detalhes proxima seção).
  - Lembrar de alterar o nome “*tabela\_observada\_pelo\_trigger*” ou “*tabela\_de\_ação\_do\_trigger*” para a forma de comunicação entre banco de dados (*nome\_database.nome\_tabela*) (ver detalhes proxima seção).
- Salvar um *backup do registro* que sofreu a ação (dados do registro).
- Salvar o *tipo do evento*, ação executada, nos registros: se foi uma inclusão (**INSERT**), modificação (**UPDATE**) ou apagamento (**DELETE**).
- No caso de uma modificação (**UPDATE**), salvar o *valor original* (**OLD.coluna**) e o *valor alterado* (**NEW.coluna**).
- Dados também muito apreciados de serem salvos no backup, dos registros, é sobre *quem fez a ação* (**CURRENT\_USER**) e o *momento em que a ação foi executada* (**NOW**).

## 17.6 Comunicação entre bancos de dado

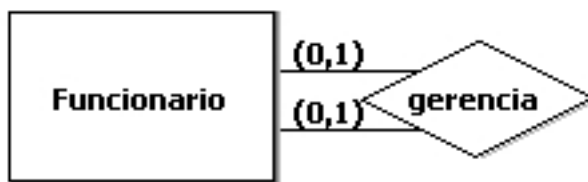
- É possível acessar dados de um **DATABASE** (*banco de dados*) estando conectado a outro **DATABASE**, sem a necessidade de fazer a mudança de **DATABASE** (**USE**).
- Ações que é possível tomar:
  - **INSERT** (*inserir registros*)
  - **SELECT** (*consulta*)
  - **DELETE** (*deletar registros*)
  - **UPDATE** (*atualizar registros*)
  - **CREATE TABLE** (*criação de tabelas*)
  - **CREATE TRIGGER** (*criação de gatilhos*)
- Para fazer tal ação ao invés de colocar o nome da tabela, usar o “nome do banco de dados” + ponto (“.”) + “nome da tabela”. Ex.: “*nome\_database.nome\_tabela*”
- Exemplo sintaxe:  
**INSERT INTO** *nome\_database.tabela* **VALUES** (...)



## 18 Módulo 20 - Autorelacionamento

### 18.1 Autorelacionamento

- Este tipo de relacionamento ocorre toda a vez que temos uma ocorrência de uma entidade que está associada a um ou mais ocorrências da mesma entidade. Ou seja, temos uma entidade onde suas ocorrências possuem relacionamentos entre si.
  - Exemplo: vamos considerar uma entidade EMPREGADO sendo que no modelo conceitual devemos representar o conceito de que um empregado possui um gerente.



- Entidade em que os atributos se relacionam.
- Cardinalidade do auto-relacionamento indica opcionalidade, se é obrigatório ou não.

### 18.2 Como construir autorelacionamento

- Basta criar uma *chave estrangeira* (**FK**) que aponte para a própria tabela.

Sintaxe:

```
CREATE TABLE tabela (  
  colunaPK INT PRIMARY KEY AUTO_INCREMENT,  
  coluna2 REGRA,  
  coluna3 REGRA,  
  colunaFK REGRA  
);
```

```
ALTER TABLE tabela  
ADD CONSTRAINT FK_colunaFK  
FOREIGN KEY (colunaFK)  
REFERENCES tabela (colunaPK);
```

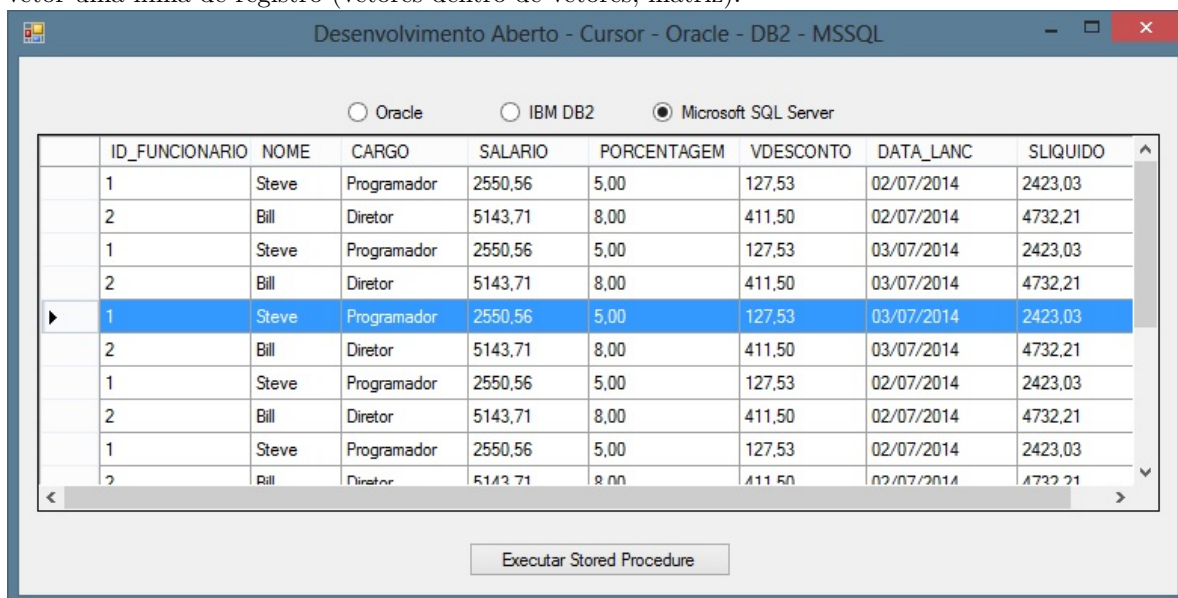
### 18.3 Retornar valor relacionado no autorelacionamento

- Atraves do **INNER JOIN** ou **LEFT JOIN** é possível retornar outro valor relacionado no autorelacionamento.
  - Ao inves de retornar um ID da *coluna chave primaria* na *coluna da chave estrangeira*, é possível retorna algum outro campo relacionado a *chave primaria* pelo valor da *coluna da chave estrangeira*.
  - Basta utilizar o **INNER JOIN** ou **LEFT JOIN**, utilizando *ponteiramento*.
- Sintaxe:  
**SELECT**  
**C.colunaPK,**  
**C.coluna2,**  
**C.coluna3,**  
**C.coluna4,**  
**IFNULL(P.coluna2, “SEM REFERENCIA”) AS REQUISITO**  
**FROM tabela C**  
**LEFT JOIN tabela P**  
**ON P.colunaPK = C.colunaFK;**
- O ponteiramento (**C** e **P**) serve para separar o que é a *tabela* e *referencia a tabela*.

## 19 Módulo 21 - Cursores

### 19.1 Teoria

- 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.
- É usado dentro de uma **PROCEDURE** (funções programadas com instruções, armazenadas pelo sistema) para realizar operações que seriam muito grandes e complicadas para um simples **SELECT**.
- 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.
- Cursores são vetores, conjunto de dados. Normalmente são usados para guardar em cada elemento do vetor uma linha de registro (vetores dentro de vetores, matriz).



	ID_FUNCIONARIO	NOME	CARGO	SALARIO	PORCENTAGEM	VDESCONTO	DATA_LANC	SLIQUIDO
	1	Steve	Programador	2550,56	5,00	127,53	02/07/2014	2423,03
	2	Bill	Diretor	5143,71	8,00	411,50	02/07/2014	4732,21
	1	Steve	Programador	2550,56	5,00	127,53	03/07/2014	2423,03
	2	Bill	Diretor	5143,71	8,00	411,50	03/07/2014	4732,21
▶	1	Steve	Programador	2550,56	5,00	127,53	03/07/2014	2423,03
	2	Bill	Diretor	5143,71	8,00	411,50	03/07/2014	4732,21
	1	Steve	Programador	2550,56	5,00	127,53	02/07/2014	2423,03
	2	Bill	Diretor	5143,71	8,00	411,50	02/07/2014	4732,21
	1	Steve	Programador	2550,56	5,00	127,53	02/07/2014	2423,03
	2	Bill	Diretor	5143,71	8,00	411,50	02/07/2014	4732,21

- Cursores vão para a memória RAM, o que leva o desempenho do servidor para baixo, mas possibilita a manipulação dos dados.
- Em resumo, o uso de **PROCEDURES** com **CURSORES** possibilita a manipulação dos registros de uma tabela, gerando assim novos campos, com os novos dados sendo o produto dessa manipulação. “Basicamente programação aplicada ao banco de dados”.

## 19.2 Principais palavras chaves

### 19.2.1 DECLARE - declaração de variaveis

- Declaração de variavel em estruturas de programação.
  - Sintaxe:  
**DECLARE** *nome\_da\_variavel tipo* [**DEFAULT** *valor*];
- Declarar varias variaveis de uma vez.
  - Sintaxe:  
**DECLARE** *variavel\_1, variavel\_2, ... tipo* [**DEFAULT** *valor*];
- Observações:
  - **DECLARE** é declaração de variavel.
  - *tipo* é o tipo da variavel (**INT**, **FLOAT**, **VARCHAR**, **CHAR**).
  - **ENUM** não é tipo, logo não pode ser declarado.
  - **DEFAULT** é um valor predefinido, é opcional.

### 19.2.2 DECLARE - declaração de variavel do tipo CURSOR

- Declara uma variavel de tipo **CURSOR**.
- Armazena dentro do **CURSOR** uma consulta (**SELECT**).
- Sintaxe:  
**DECLARE** *nome\_da\_variavel\_CURSOR* **CURSOR FOR** (  
**SELECT**  
*coluna1,*  
*coluna2,*  
*... FROM tabela*  
);
- Observação:
  - Não leva “;” ao final da consulta, porem leva no fechamento do “()”.
  - As colunas/campos ficam armazenada no **CURSOR**, na ordem em que são listadas na consulta.
  - Não confundir, cada linha de registro (com “n” colunas/campos) é **UM** elemento do vetor **CURSOR**.

### 19.2.3 OPEN e CLOSE - manipulação de variáveis

- **OPEN**

- Leva a variável do tipo **CURSOR** para a memória RAM para poder ser manipulada.
- Usado antes do **REPEAT**.
- Sintaxe:  
**OPEN** *nome\_da\_variavel\_CURSOR*;

- **CLOSE**

- Fecha a variável do tipo **CURSOR**, remove da memória RAM.
- Comumente usado depois do **REPEAT**.
- Sintaxe:  
**CLOSE** *nome\_da\_variavel\_CURSOR*;

#### 19.2.4 DECLARE CONTINUE HANDLER - declarando variavel de manipulação continua

- Declaração de um robô que observa os elementos do vetor **CURSOR** no loop (**REPEAT**).
- Quando os elementos do vetor acabam, modifica uma variavel que serve como criterio de parada para o loop.
- Declarado antes do loop.
- Sintaxe:  
**DECLARE *FIM* INT DEFAULT 0;**  
**DECLARE CONTINUE HANDLER FOR NOT FOUND SET *FIM* = 1;**
- Observações:
  - “Formula de bolo”, sempre se repete escrito desta forma.
  - Depois que percorre todos os elementos, passa o valor “**NOT FOUND**” (não encontrado) e modifica a variavel *FIM* para valor 1.
  - Variavel *FIM* pois finaliza o loop, nome dado a uma varaivel qualquer.

### 19.2.5 IF

- **IF**

- Tomador de decisão simples.

- Sintaxe:

- IF** *condição* **THEN**  
[*bloco de programação SQL*];  
**END IF**;

- **IF e ELSE**

- Caso o tomador de decisão **IF** falhe, o **ELSE** deve ser executado.

- Sintaxe:

- IF** *condição* **THEN**  
[*bloco de programação SQL*];  
**ELSE**  
[*bloco de programação SQL*];  
**END IF**;

- **IF, ELSEIF e ELSE**

- Varios casos de decisões (**ELSEIF**) alem do **IF**, caso o **IF** falhe.

- Caso todos falhem (**IF** e **ELSEIF**), o **ELSE** deve ser executado.

- Sintaxe:

- IF** *condição1* **THEN**  
[*bloco de programação SQL*];  
**ELSEIF** *condição2* **THEN**  
[*bloco de programação SQL*];  
**ELSE**  
[*bloco de programação SQL*];  
**END IF**;

### 19.2.6 REPEAT - Loop

- Faz um loop que se repete ate determina expressão seja verdadeira.
- O uso de **REPEAT** é otimo para fazer operações (manipulação de dados), linha por linha de um determinada tabela de um banco de dados, gerando novos dados derivados.
- Sintaxe:  
**REPEAT**  
*[bloco de programação em SQL];*  
**UNTIL** *expressão*  
**END REPEAT;**
- Observações:
  - O bloco de programação em SQL, pode conter o **FETCH** para percorrer um **CURSOR**.
  - Para finalizar um **REPEAT**, podemos programar (declarar) um **CONTINUE HANDLER** antes do loop, e substituir a *expressão* por “*FIM = 1*”, para sair do loop depois que o **CURSOR** (vetor) for todo percorrido, com ajuda do **FETCH**.
  - É comum usar os comandos SQL (**INSERT**, **UPDATE** e **DELETE**) dentro do bloco de programação.
  - Em especial o **INSERT** é util para gravar os dados, novos e/ou modificados, num banco de dados.
  - Outro comando que se mostra util é o uso de **IF**.

### 19.2.7 FETCH - chama o proximo elemento do CURSOR no Loop

- Funciona dentro do loop.
- Chama o proximo elemento do **CURSOR**, começando do elemento 1.
- Vai percorrendo o **CURSOR** a cada loop, 1 elemento do **CURSOR** por loop.
- Sintaxe:  
**FETCH** *nome\_da\_variavel\_CURSOR* **INTO** *variavel1, variavel2, ...;*
- Observações:
  - As variaveis devem estar previamente declaradas.
  - O **FETCH** adiciona o valor dos campos do elemento do **CURSOR** em cada variavel, na ordem em que os campos foram chamados na declaração do **CURSOR**.
  - Apartir do **FETCH**, pode-se trabalhar com as variaveis pois elas vão estar com o valor de cada campo, de cada linha de registro a cada looping.



### 19.3 Juntando tudo - **CURSOR**

- **CURSOR**, assim como vetores, são usados para guardar registros para percorrer um determinado tabela de banco de dados.
- Lembrando que **CURSOR** é normalmente usado dentro de **PROCEDURE**.
- Por conta disso, lembrando de mudar o **DELIMITER** antes e depois do **PROCEDURE**.
- Para chamar o **PROCEDURE**, utilizar o **CALL**.
- Juntando tudo que foi estudado para **CURSOR**:
  - **DECLARE**
  - **DECLARE CURSOR**
  - **OPEN** e **CLOSE**
  - **CONTINUE HANDLER**
  - **REPEAT**
  - **FETCH**

- Sintaxe:
 

```

DELIMITER #
CREATE PROCEDURE nome_do_procedure()
BEGIN
DECLARE FIM INT DEFAULT 0;
DECLARE variavel1, variavel2, ..., variaveln INT;
DECLARE variavel_nome VARCHAR(50);
DECLARE variavel_nome_do_cursor CURSOR FOR (
SELECT
coluna1,
coluna2,
coluna3,
coluna4,
coluna5
FROM tabela
);
DECLARE CONTINUE HANDLER FOR NOT FOUND SET FIM = 1;
OPEN variavel_nome_do_cursor;
REPEAT
FETCH variavel_nome_do_cursor INTO variavel1, variavel_nome, variavel2, variavel3, variavel4;
IF NOT FIM THEN
[exemplo de bloco de programação SQL]
SET variavel5 = variavel1 + variavel2 + variavel3;
SET variavel6 = variavel5 / 3;
INSERT INTO tabela_nova VALUES
(NULL, variavel1, variavel_nome, variavel2, variavel3, variavel4, variavel5, variavel6);
END IF;
UNTIL FIM END REPEAT;
CLOSE variavel_nome_do_cursor;
END
#
DELIMITER ;
CALL nome_do_procedure();
      
```
- Observações:
  - **SET** é para atribuir valores.
  - Em declaração de **CURSOR**, o “;” só vai no fechando paranteses “()”.
  - Antes de fazer a **PROCEDURE**, é necessario preparar uma nova tabela no banco de dados para receber os novos valores.

## 20 Módulo 23 - Introdução a Business Intelligence

### 20.1 Banco de dados relacional

- Foco:
  - Rotinas do dia-dia do negócio.
- Principais rotinas:
  - **INSERT**  
Inserir dados.
  - **UPDATE**  
Modificar dados.
  - **DELETE**  
Excluir dados.
- Modelagem:
  - 1º forma normal
  - 2º forma normal
  - 3º forma normal
- Por que modelar?
  - Evitar redundancia de dados.
  - Separar dados por diversas tabelas.
  - Por consequencia, evitar que o HD cresça.
- Consequencias:
  - Pouca eficiência nas consultas.
  - Devido a diversas junções de tabelas (**JOIN**) que aumenta o desempenho da maquina para as consultas.

### 20.2 Business Intelligence

- Foco:
  - Em consultas (**SELECT**).
- Modelagem:
  - Desnormalizar os dados, para aumentar a eficiencia das consultas.
- Por que desnormalizar?

- Aumentar a eficiencia das consultas, sem se preocupar (muito) com o espaço ocupado por esses dados.
  - Gerar consultas rapidas aos dados, para apoiar os diversos *stakeholders* nas tomadas de decisões do negócio.
- Consequencias:
  - Gera redundancia de dados, aumenta o espaço de armazenamento.
  - Diminui o desempenho das maquinas para as consultas, tornando o processo de consulta mais eficiente.

## 21 Detalhes

- **Comentarios** no **MySQL**, diferente do **SQL** onde comentarios são `'/**/`, no **MySQL** é `#`. Ou `'- '` para comentario de linha.
- O que são e o que fazem os administradores:
  - Administrador de dados(AD):

O Administrador de Dados (AD) tem o objetivo de gerenciar o Modelo de Dados Corporativo, contribuindo para assegurar a qualidade das informações, a integração dos sistemas, a retenção e a disseminação do conhecimento dos negócios.

Cabe a ele, guiado por certos princípios e através de atividades de planejamento, organização e controle dos dados corporativos, gerenciar os dados como recursos de uso comum da organização, promovendo-lhes os valores de autenticidade, autoridade, precisão, acessibilidade, seguridade e inteligibilidade.

Tem como função o planejamento central, a documentação e o gerenciamento dos dados a partir da perspectiva de seus significados e valores para a organização como um todo.
  - Administrador de banco de dados (DBA):

O DBA (database administrator), sigla em inglês para Administrador de Banco de Dados, é um profissional da área de tecnologia responsável pela criação, instalação, monitoramento, reparos e análise de estruturas de um banco de dados.

O banco de dados fica sob análise periódica do DBA, que trabalha para que não haja sobrecargas do sistema e que as informações inseridas tenham destino correto nos servidores. Outras funções também importantes são analisar o espaço em disco, buscar melhorias para os sistemas e realizar backups.
- Acesso ao **MySQL** pelo terminal é necessario usar o comando:  
`mysql -u root -p`
  - Depois colocar a senha.
- Ao final dos comandos do **SQL** e do **MySQL**, usar o `;`(delimitador), ele informa que o comando acabou e deve ser executado.
- O **MySQL** é “*case sensitive*” no **LINUX**, mas no **WINDOWS** não é, ou seja, sensibilidade a letras maiúsculas e minúsculas. Depende do sistema operacional. **MySQL** acompanha o sistema operacional.
  - no **LINUX** faz distinção de letras maiúsculas e minúsculas.
  - no **WINDOWS** não faz distinção de letras maiúsculas e minúsculas.

## **22 Andamento dos Estudos**

### **22.1 Assunto em andamento:**

Curso concluído.