

SQL Completo

[Apostila baseada no curso de mesmo nome da Softblue]

[Essa apostila complementa a apostila do Curso em Vídeo]

1. Introdução

1.1. O que são banco de dados

Mecanismos de armazenamento de dados que, por sua vez, são conjuntos de informações com uma estrutura regular, podendo ser padronizada.

Tipos de banco de dados:

Bancos de dados não relacionais: utilizam um formato regular para escrever as informações sequenciais em um arquivo estruturado, sendo que, o acesso às informações, se torna mais lento.

Bancos de dados relacionais: permitem que os dados sejam organizados em tabelas que, por sua vez, se relacionam entre si. Esses bancos também ocupam um menor espaço de armazenamento, o que dá uma maior velocidade de acesso a esses dados. Por fim, uma das características mais importantes é o fato de ser um padrão mundialmente utilizado.

1.1.1. Banco de dados relacionais

Ferramentas que permitem o armazenamento e manipulação de dados organizados em forma de tabelas.

1.1.1.1. Alguns conceitos

Tabelas: É uma forma de organização de dados, sendo constituída por linhas e colunas;

Colunas: Informações que desejamos armazenar, também conhecidas como campos que formam um registro;

Linhas: Registros armazenados nas tabelas, também conhecidos como tuplas;

Visões: Consultas SQL previamente programadas, disponíveis para um rápido acesso. Além disso, as visões facilitam a manutenção do banco de dados. Elas não armazenam dados, mas sim critérios de seleção dos mesmos (critérios de busca), podem ser reutilizadas sempre que as tabelas **sofrerem alteração, exclusão ou são criadas**. Os dados usados nas visões são dinâmicos;

Índices: Estruturas que gerenciam a ordenação dos registros (valores) dos campos desejados para melhorar a performance do processamento destes campos.

Banco de dados

Tabela PRODUTOS

PRODUTO	PREÇO
TV	1000
DVD Player	290
Blu-ray Player	490

Tabela CLIENTES

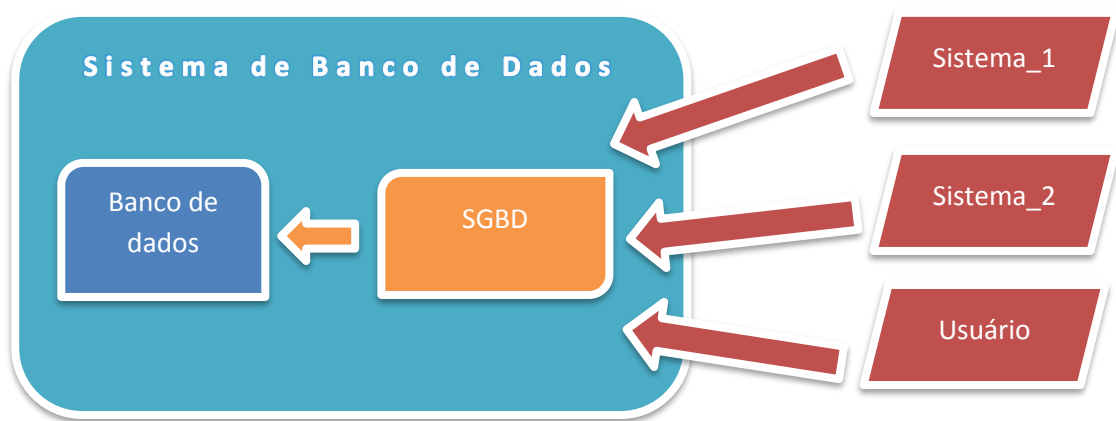
NOME	IDADE	ESTADO
Fernando	25	PR
Guilherme	30	SP
Mateus	35	RJ

1.2. SGBD (Sistema Gerenciador de Banco de Dados)

Do inglês DBMS (**D**atabase **M**anagement **S**ystem), são ferramentas que auxiliam na manipulação e organização dos bancos de dados. Por exemplo, a funcionalidade de armazenar uma informação, alterá-la ou recuperá-la é do banco de dados, mas o gerenciamento de acesso ao banco é do SGBD.

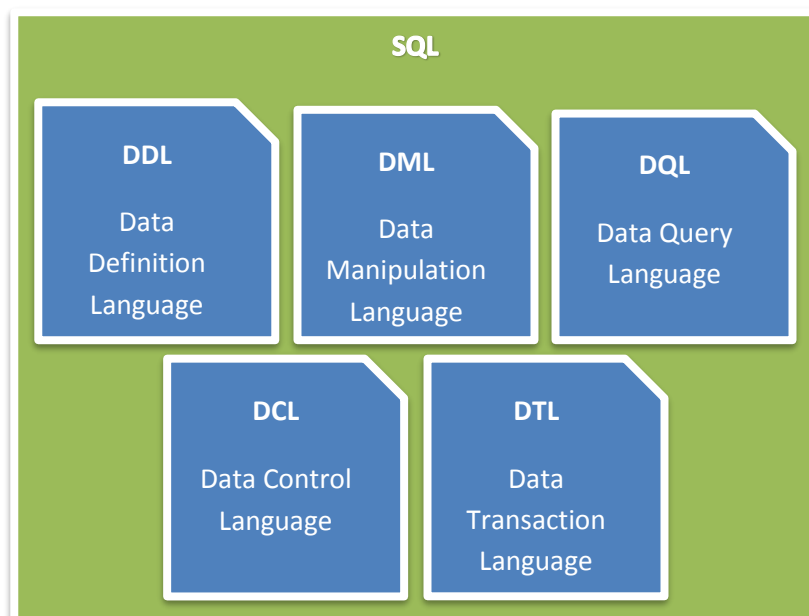
Eles são usados em aplicações que farão uma grande quantidade de **acesso** e operações simultâneas no banco de dados, evitando a ocorrência de inconsistências no mesmo (a venda de um mesmo carro para duas pessoas que estão tentando comprá-lo ao mesmo tempo, por exemplo).

Resumindo, o **SGBS** não é um banco de dados, mas um complemento a este, sendo um grupo de programas de interação com dados.



1.3. SQL (Structured Query Language)

Em português, Linguagem de Consulta Estruturada, responsável pela interação com os bancos de dados relacionais. Essa linguagem se diferencia das outras linguagens de consulta de banco de dados por gerar uma consulta SQL específica para os resultados que se deseja dos dados (se estrutura os dados que se quer obter no formato e na ordem desejada). Ela é formada por um conjunto de linguagens:



1.3.1. DDL – Data Definition Language (Linguagem de Definição de Dados)

Permite definir a estrutura do banco como tabelas, visões, índices, sequências e etc.

Principais comandos:

CREATE: Cria uma estrutura
ALTER: Altera uma estrutura
DROP: Exclui uma estrutura*

1.3.2. DML – Data Manipulation Language (Linguagem de Manipulação de Dados)

Uma vez definido o formato dos dados, a DML permite manipular (gerenciar) os mesmos.

Principais comandos:

INSERT: Insere dados
UPDATE: Altera dados
DELETE: Exclui dados*

* A diferença entre o *delete* e o *drop* é que, o primeiro exclui um determinado dado (registro), já o segundo exclui toda uma estrutura que faz parte do banco de dados.

1.3.3. DQL – Data Query Language (Linguagem de Consulta de Dados);

Permite recuperação e leitura dos dados inseridos nas tabelas e nas estruturas do banco de dados.

Principal comando:

SELECT: Retorna, ordena, agrupa dados (por algum parâmetro), faz cálculo de operações matemáticas e aplica de filtros de seleção.

1.3.4. DCL – Data Control Language (Linguagem de Controle de Dados);

Permite gerenciar quem pode ou não realizar determinada tarefa (ação ou operação) dentro do banco de dados.

Principais comandos:

CREATE USER: Cria um usuário

DROP USER: Exclui um usuário

GRANT: Concede acesso a dados e operações

REVOKE: Revoga acesso a dados e operações

1.3.5. DTL – Data Transaction Language (Linguagem de Transação de Dados).

Gerencia um conjunto operações SQL que, juntas, formam uma determinada operação SQL mais complexa.

Principais comandos:

START TRANSACTION: Inicia uma transação

COMMIT: Concretiza uma transação

ROLLBACK: Anula (com segurança) uma transação

	MySQL	PostgreSQL	Firebird	Oracle	SQL Server
SGBD	Sim	Sim	Sim	Sim	Sim
ACID (transações poderosas)	Sim	Sim	Sim	Sim	Sim
Licença Comercial	Não	Sim	Sim	Não	Não
Licença Estudante	Sim	Sim	Sim	Oracle Express	SQL Server Express

2. Normalização dos dados

É a separação das informações de um banco de dados (“tabelas aninhadas”) em outras tabelas separadas, mas correlacionadas. Principais vantagens:

- Evita anomalias (aninhamento de tabelas, redundância de dados e inconsistência no banco de dados);
- Facilita a manutenção;
- Maximiza a performance (devido à disposição das informações);
- Mantem a integridade dos dados.

2.1. Relacionamento e chaves

Relacionamento são ligações entre tabelas, se dá por meio da existência de um ou mais campos comuns (campos chaves) entre as mesmas, servindo assim, como chaves de identificação.

2.1.1. Tipos de chaves

Chave primária (Primary Key, PK): Quando uma determinada coluna não possui valores que se repedem nos registros, também conhecida como identificador único;

Chave composta: Caracterizada quando há necessidade de duas ou mais colunas (campos) para caracterizar o identificado único, isso acontece quando uma tabela não possui nenhum campo que, isoladamente, pode se tornar uma chave primária.

Chave estrangeira (Foreign Key, FK): Quando uma coluna é utilizada em uma tabela apenas para fazer referência a uma chave primaria de outra ou da mesma tabela, ela é considerada uma chave estrangeira. Sendo assim, valor da chave estrangeira deve também existir na tabela de origem do registro.

Todas as chaves podem receber valores que não sejam numéricos, também.

Boas práticas no uso de chaves:

- Quanto menor, melhor: Os tipos dos campos devem ser os menores possíveis, gerando uma economia de espaço de armazenamento no computador e facilitando seu processamento pelo computador;
- Transparentes para o usuário: Utilização de códigos próprios (internos) evitando erros no banco de dados e transtorno para o cliente.

2.1.2. Tipos de relacionamentos e seus respectivos modelos de dados

Relacionamento um para um (1:1): Para cada registro da primeira tabela devemos ter, no máximo, um correspondente na segunda tabela, e vice-versa.



Relacionamento um para muitos (1:*): Para cada registro da primeira tabela pode haver um ou mais registro relacionado na segunda tabela, mas a segunda tabela pode ter somente um registro correspondente com a primeira.



Relacionamento muitos para muitos (*:*): Cada registro na primeira tabela pode se corresponder com um ou mais registros da segunda, e vice-versa.



2.2. Anomalias dos dados

As tabelas que armazenam muitas informações variadas (tabelas “fazem tudo”) geram anomalias.

2.2.1. Principais anomalias

Anomalia de inserção: Impede a inclusão de registros no banco de dados devido à falta de dados. Pode ser evitada criando uma tabela independente, contendo informações que podem ser independentes.

Anomalia de exclusão: Impede a exclusão de registros devido ao relacionamento com outra tabela.

Anomalia de alteração: Impede a alteração de registros devido ao relacionamento com outra tabela. Pode ser evitado criando uma chave interna dentro do banco de dados.

2.3. Formas normais (FNs)

Passos que descrevem o que deve ser feito para organizar os dados do banco da melhor forma possível. Isso implica no aumento do número de tabelas, devido ao desmembramento, conseqüentemente aumenta a manutenção e diminui a performance, mas somente a curto prazo. Devido a esses imprevistos, a normalização deve ser usada com bom senso.

1ª Forma Normal (1FN)

- Cada linha do banco de dados deve representar apenas um registro;
- Cada célula/campo da tabela deve conter um único valor.

2ª Forma Normal (2FN)

- Obrigatoriamente estar na 1FN;
- Atributos não chave da tabela devem depender de alguma das chaves da tabela.

3ª Forma Normal (3FN)

- Obrigatoriamente estar na 2FN;
- Atributos não chave da tabela devem depender exclusivamente da chave primária da tabela.

4ª e 5ª Forma Normal (4FN E 5FN)

- Separam em novas tabelas valores que ainda estejam redundantes em uma mesma coluna.

3. Criando um banco de dados

3.1. Tipos de dados

Os tipos de dados aceitos em cada banco de dados variam, mas os tipos mais comuns estão presentes na tabela “Tipos de dados comuns em bancos de dados”.

OBS: Os valores reais aceitam receber valores do tipo *string*, embora não seja muito recomendada essa prática.

Além dos tipos de dados presentes na tabela citada, temos ainda:

Blob: Que permite o armazenamento de informações binárias, arquivos e imagens, sendo necessária a atuação de uma linguagem de programação para torna-los “legíveis” de novo ou armazená-los no banco de dados;

Text: Permite o armazenamento de grandes informações em texto (artigos, livros, etc.).

Redes: Permite o armazenamento de endereços IP, MAC-ADDRESS e outros.

Monetários: Permite o armazenamento de valores monetários com formatação (reais, euros, etc.)

Geométricos: Permite o armazenamento de informações de formas geométricas (linhas, quadrados, círculos, etc.).

3.2. Atributos

São parâmetros que podem ser adicionados na criação dos campos de uma tabela e que permitem especificar o tipo de informação que os mesmos podem receber.

null / not null: Permite ou não valores nulos.

signed / unsigned: Permite ou não valores negativos. Se tratando do tipo de dado inteiro, quando recebe a propriedade *unsigned* (sem sinal) o intervalo positivo desse valor é dobrado.

auto-increment: Adiciona de forma automática (pelo próprio banco de dados) uma determinada quantidade de valor toda vez que um novo registro é gerado. Esse atributo geralmente é usado em chaves primárias.

zerofill: Preenche o valor numérico completando com zeros a esquerda

primary key: Torna um determinado campo uma chave primária de uma tabela

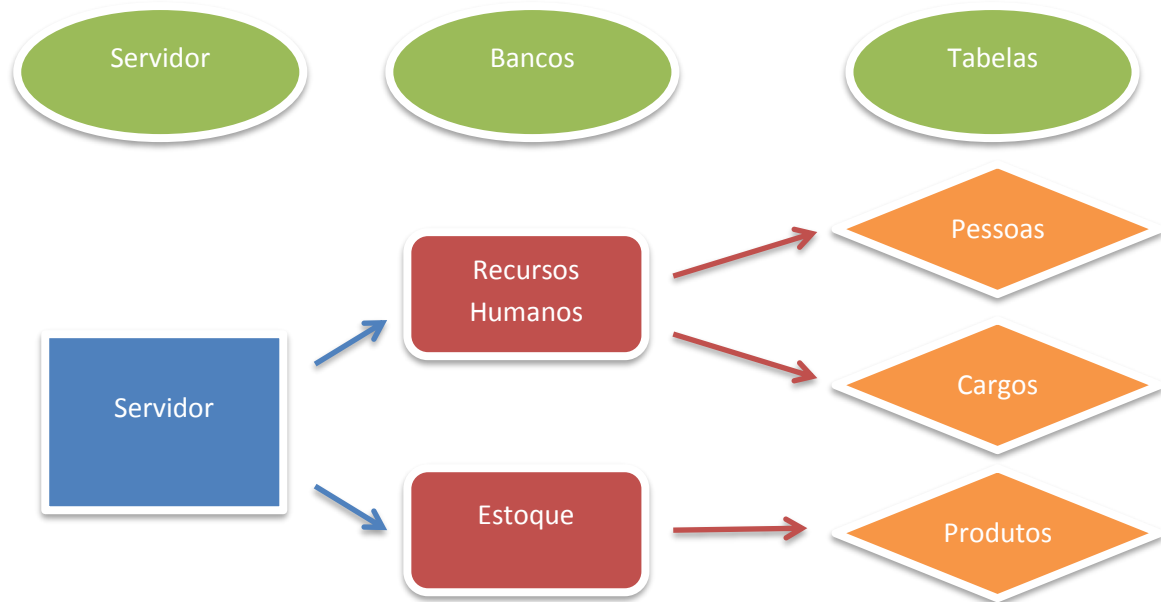
default: É um valor usado como padrão em uma tabela, caso o usuário não informe algum.

3.3. Boas práticas de armazenamento

- Quanto menor o tipo de dado, menos espaço ele ocupará, isso se deve ao fato de que, certos subtipos de dados, são preenchidos por um determinado valor pelo próprio banco de dados, para ocupar o que não foi usado. Deve se atentar ao tamanho do dado e o limite do subtipo escolhido para seu respectivo campo. Essa regra se aplica a todas as categorias de dados e deve responder a seguinte pergunta, “Qual o menor e maior valor que o campo poderá receber?”.
- Quanto menor o tipo de dado, mais rápido é o processamento, isso porque não terá muitas informações grandes a serem verificadas.
- Não armazenar dados *string* em colunas de valor numérico, principalmente se esses números farão parte de algum cálculo, evitando, assim, a necessidade de conversão dos mesmos.

3.4. Estrutura básica da transição de uma informação

Um servidor pode ter mais de um banco de dados.



3.5. Comandos (DDL) e regras de definição de bancos de dados

- O nome do banco de dados não deve conter espaços e nem caracteres especiais, com exceção do _.
- Os conjuntos de caracteres especiais mais utilizados são Latin1 e UTF-8
- O comando *drop* exclui definitivamente uma estrutura.

Exemplo dos principais comandos:

```
CREATE DATABASE [NOME];  
ALTER DATABASE [NOME] [PROPRIEDADE];  
DROP DATABASE [NOME];  
USE [BANCO DE DADOS];*
```

* Usado para indicar o banco de dados que será usado (manipulado) em um determinado momento.

3.5.1. Criação de uma tabela

Principais comandos:

```
CREATE TABLE [NOME] ([CAMPOS, TIPOS DOS DADOS, ATRIBUTOS]);  
ALTER TABLE [NOME] [PROPRIEDADE];  
DROP TABLE [NOME];
```


Na criação de uma chave estrangeira (Foreign Key, FK), o campo que fará referência a mesma deve ser do mesmo subtipo e ter a propriedade de *unsigned*, caso possua, da chave a qual ela faz referência.

constraint ["APELIDO"] foreign key([NOVA CHAVE]) references [TABELA DA PRIMÁRIA]([CHAVE PRIMÁRIA]);

Exemplo:

```
CREATE TABLE funcionarios (  
    id int unsigned not null auto_increment,  
    nome varchar(45) not null,  
    salarios double not null default "0",  
    departamento varchar(45) not null,  
    primary key(id)  
);
```

```
CREATE TABLE veiculos (  
    id int unsigned not null auto_increment,  
    funcionario_id int unsigned,  
    veiculo varchar(45) not null default "",  
    placa varchar(10) not null default "",  
    primary key(id),  
    constraint fk_veiculos_funcionarios  
    foreign key(funcionario_id) references  
    funcionarios(id)  
);
```



3.5.2. Índices

Os índices são estruturas independentes do banco de dados, geralmente algum campo, que armazenam de forma isolada o ordenamento dos registros de uma tabela específica, baseando-se no índice em que foi criada. Essas estruturas sofrem alterações de acordo com alterações feitas na tabela de origem.

Elas geralmente são criadas quando há muita requisição a uma determinada informação da tabela ou quando há muitos registros a serem processados todas as vezes que for requisitada tal ordenação. Só exibirá aquilo que foi determinado na sua criação.

Deve-se criar os índices somente para os campos mais usados, economizando espaço em disco e processamento.

Exemplos dos principais comandos:

```
CREATE INDEX [NOME] ON [TABELA]([COLUMA]) (CARACTERES) OPCIONAL |);  
ALTER INDEX [NOME] [PROPRIEDADE];  
DROP INDEX [NOME] ON [TABELA];*
```

* O *drop*, nesse caso, não altera a estrutura da tabela, somente exclui um determinado índice.

3.5.3. Criando uma sequência

É um número que, sempre que for utilizado, será incrementado com um determinado número definido na sua criação.

Exemplo:

```
CREATE SEQUENCE [NOME];  
ALTER SEQUENCE [NOME] [PROPRIEDADE];  
DROP SEQUENCE [NOME];
```

4. Manipulando e consultando dados

As linguagens responsáveis pela manipulação e consulta são a Data Manipulation Language e a Data Query Language, respectivamente.

Alguns comandos DML (update e o delete, especificamente), bem como o *select* da DQL, são comandos gerais que alteram ou consultam todos os registros de uma tabela, sendo necessário um filtro para especificar o que deseja mudar em apenas um registro. Dentre os filtros de seleção mais usados, estão:

where: Permite que condições sejam especificadas, para que uma determinada consulta ou manipulação de registros venha a ser feita. Essas condições são especificadas através do uso os operadores relacionais (=, !=, >, >=, <, <=, *is null*, *is not null*, *between*, *like*) e operadores lógicos (*and*, *or*, *not*).

4.1. Atributos especiais

Apelido de tabela: Serve para facilitar a escrita do comando, quando se tem muitas referências a tabela em questão. Úteis quando se tem tabelas com campos de mesmo nome, então se usa um apelido para distingui-las.

A sintaxe para dar apelido a um campo é:

```
SELECT [CAMPO] AS [APELIDO] FROM [TABELA];
```

Para usar esse campo:

```
SELECT [APELIDO].[CAMPO] FROM [TABELA] [APELIDO];
```

OBS: Esse apelido pode conter espaços, desde que ele esteja dentro de aspas.

União de seleções (union ou union all): Usado para juntar duas expressões *select*, retornando uma única tabela, como resultado, do banco de dados.

```
SELECT [CAMPOS_1] FROM [TABELA_1] UNION SELECT [CAMPOS_2] FROM [TABELA_2];
```

O *union* não retornará registros duplicados, caso ambas as tabelas tenham registros em comum, diferentemente do *union all*, que retornará o número de vezes que cada registro aparece em cada uma das expressões *select* (mesmo que eles já tenham aparecido). Os retornos das duas expressões *select* devem ser as mesmas.

Exemplo do *union*:

```
38 • SELECT * FROM funcionarios WHERE nome = 'Guilherme'
39 UNION
40 SELECT * FROM funcionarios WHERE id = 5;
```

id	nome	salario	departamento
2	Guilherme	3025	Jurídico
5	Isabela	2662	Jurídico

Exemplo do *union all*:

```
42 • SELECT * FROM funcionarios WHERE nome = 'Guilherme'
43 UNION ALL
44 SELECT * FROM funcionarios WHERE nome = 'Guilherme';
45
46
```

id	nome	salario	departamento
2	Guilherme	3025	Jurídico
2	Guilherme	3025	Jurídico

O comando `SAFE UPDATES` impede que haja uma atualização geral em todo um campo da tabela de uma única vez, para desabilitar essa opção, faça:

```
SET sql_safe_updates = 0;
```

Caso queira informar ou alterar um número real com uma quantidade de casas decimais definidas, pode-se utilizar a função *round*.

`round([VALOR], [CASAS])`

5. Relacionamento e visões

5.1. Relacionamento

Relacionamentos são informações relacionadas entre si, geralmente com campos comuns que, por sua vez, geralmente são campos chaves, mas que estão em tabelas diferentes ou na mesma. Essa relação pode se dar entre mais de duas tabelas.

O parâmetro *join* apresenta várias variações e, embora a maioria esteja implementada nos bancos de dados existentes, pode ocorrer algumas incompatibilidades, sendo necessário consultar o manual desses mesmos bancos de dados.

Inner join: Também conhecido como *join* padrão, ele é usado em relacionamentos onde não há outro *join* especificado, combinando registros da primeira tabela com registros da segunda, desde que satisfaçam as condições do comando *join* previamente definidos.

Exemplo:

```
SELECT * FROM pessoas INNER JOIN veiculos ON pessoas.cpf = veiculos.cpf;
```

Exemplo com mais de duas tabelas:

```
SELECT * FROM pessoas INNER JOIN veiculos ON pessoas.cpf = veiculos.cpf  
INNER JOIN ipva ON veiculos.ipva = ipva.id;
```

Não há necessidade de informar os nomes das tabelas depois do *on* se as colunas a serem mostradas possuem nomes diferentes entre ambas as tabelas (não tendo uma ambiguidade).

Equi join: Similar ao *inner join*, com a diferença de que o nome dos campos entre as tabelas deve ser o mesmo. Na tabela resultado, aparecerá o apenas

uma vez o nome do campo, uma vez que ele é igual, diferentemente do que acontece no *inner join*.

Exemplo:

```
SELECT * FROM pessoas INNER JOIN veiculos USING (cpf);
```

Non equi join: Relacionamento entre tabelas que não possui campos em comum. Esse comando permite o uso de apelidos para designar tabelas.

Exemplo:

```
SELECT p.nome, p.salario, s.faixa FROM pessoas p INNER JOIN salarios s ON p.salario BETWEEN s.inicio AND s.fim;
```

Outer join, left join ou left outer join: Relacionamento que trará como resultado os registros do primeiro (left) campo, além do segundo, mesmo que eles não tenham correspondentes com o segundo campo (não satisfazendo a regra).

Exemplo:

```
SELECT * FROM pessoas LEFT JOIN veiculos ON pessoas.cpf = veiculos.cpf;
```

Right join, right outer join: Diferentemente da *left outer join*, é a segunda tabela cujo campo de condição não satisfaça a união de tabelas, mostrando assim os registros que não tem relação com o campo da primeira tabela.

Exemplo:

```
SELECT * FROM pessoas RIGHT JOIN veiculos ON pessoas.cpf = veiculos.cpf;
```

Full outer join: É a combinação de *left join* e *right join*. Linhas da primeira e segunda tabela cujos campos de condição não satisfaçam a união de tabelas. O MySQL NÃO TEM esse comando, mas pode-se chegar a esse mesmo resultado fazendo o uso do comando *union* entre o *left* e o *right join*.

Exemplo em outros Banco de Dados:

```
SELECT * FROM pessoas FULL JOIN veiculos ON pessoas.cpf = veiculos.cpf;
```

Exemplo no MySQL:

```
SELECT * FROM pessoas LEFT JOIN veiculos ON pessoas.cpf = veiculos.cpf UNION  
SELECT * FROM pessoas RIGHT JOIN veiculos ON pessoas.cpf = veiculo.cpf;
```

Self join: Une uma tabela com ela mesma. Nesse tipo de relacionamento há a necessidade do uso de apelidos.

Exemplo:

```
SELECT a.nome, b.nome AS "indicado por" FROM pessoas a JOIN pessoas b ON  
a.indicado = b.cpf;
```

5.2. Visões

São tabelas virtuais ou relações virtuais pré-programadas que não armazenam dados e sim retornam o resultado de uma consulta SQL específica, não fazendo parte do modelo lógico, podendo ser excluída sem perder nenhuma informação do banco.

A expressão SQL que monta uma visão fica armazenada no banco de dados e pode ser alterada para atender novos requisitos de consulta sem a necessidade de criar uma nova visão, centralizando o código e facilitando a manutenção da expressão SQL, ou seja, pode ser reutilizada.

Sua sintaxe:

```
CREATE VIEW [NOME] AS [EXPRESSÃO SQL];  
  
ALTER VIEW [NOME] [NOVA EXPRESSÃO OU PROPRIEDADE];  
  
DROP VIEW [NOME];
```

Para executar uma *view* já pronta:

```
SELECT * FROM [NOME DA VIEW];
```

6. Funções especiais e subqueries

6.1. Funções especiais

Funções de paginação no SQL: Mostra uma quantidade definida de registros de uma seleção (de cem mostra apenas os vinte primeiros, por exemplo).

Entre os parâmetros suportados pela paginação, estão:

- *distinct* que permite mostrar somente os registros que não se repetem.

Exemplo:

```
SELECT DISTINCT (departamento) FROM funcionarios;
```

- *limit*: limite a quantidade de registros que serão retornados.

Exemplo:

```
SELECT * FROM funcionarios LIMIT 2;
```

- *offset* especifica a quantidade de registros que serão avançados (“pulados”).

Exemplo:

```
SELECT * FROM funcionarios LIMIT 2 OFFSET 2;
```

ou

```
SELECT * FROM funcionarios LIMIT 2, 2;
```

OBS: No *offset* utilizando vírgula, primeiro vem o valor do *offset*, depois a quantidade a ser mostrada (*limit*).

6.2. Subqueries

Possibilita a realização de uma consulta com filtros de seleção baseado em outra lista ou seleção.

Tem, como parâmetros, o *in* e o *not in*, que inclui ou exclui, respectivamente, registro em uma determinada consulta. Eles também podem receber o resultado de outras consultas.

Exemplo:

```
SELECT nome FROM funcionarios WHERE departamento IN (SELECT departamento FROM funcionarios GROUP BY departamento HAVING AVG(SALARIO > 1500);
```

7. Controle de acesso

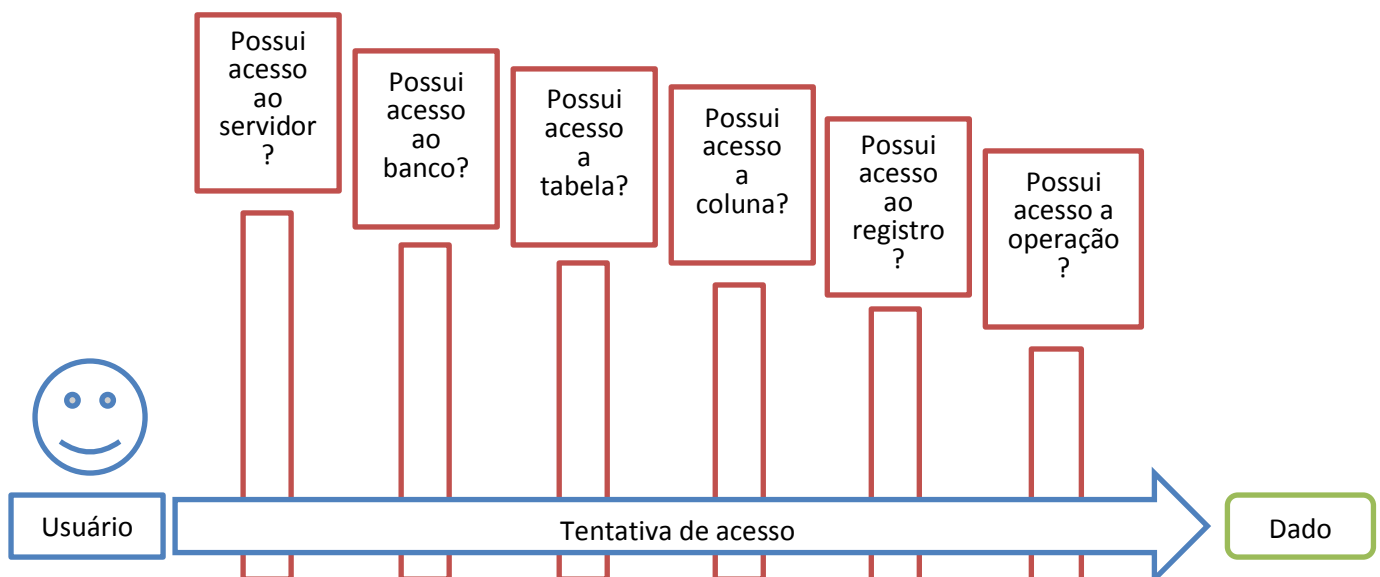
A linguagem DCL (Data Control Language) é responsável pelo controle de dados, garantindo que somente pessoas ou softwares autorizados possam acessar ou fazer operações específicas no banco de dados.

Existem vários níveis de acesso ao banco de dados:

- Todo o banco de dados;
- A tabelas específicas;
- A colunas específicas;
- A registros específicos.

As ações que podem ser realizadas na estrutura da tabela, são:

- Gerenciamento da estrutura;
- Gerenciamento dos dados;
- Leitura de dados.



Os bancos de dados presentes no mercado possuem diferentes tipos/níveis de acesso, sendo necessário consultar seus respectivos manuais para mais informações.

Os comandos da Linguagem de Controle de Dados mais usados são:

CREATE USER "[NOME]"@"[LOCAL DE ACESSO]" **IDENTIFIED BY** "[SENHA]": Cria um usuário
DROP USER "[NOME]": Exclui um usuário
GRANT [AÇÃO] **ON** [ESTRUTURA] **TO** "[USUÁRIO]"@"[LOCAL DE ACESSO]": Habilita acessos
REVOKE [AÇÃO] **ON** [ESTRUTURA] **FROM** "[USUÁRIO]"@"[LOCAL DE ACESSO]": Revoga acessos

Dentro das [AÇÕES] podemos colocar os comandos:

- *All* : Dá total controle de manipulação ao usuário;
- *Select* : Dá somente a permissão de leitura dos dados pelos registros para o usuário;
- *Insert* : Permite a inserção de registro na tabela;
- *Update* : Permite somente a modificação de valores;
- *Delete* : Permite somente a exclusão de valores.

Esses comandos têm a ver com o nível de acesso dado a um usuário.

Em [LOCAL DE ACESSO] diz respeito à forma como o usuário terá acesso ao banco de dados, se é por um IP fixo, qualquer (através do símbolo "%"), do próprio servidor ("*localhost*"). O usuário ou estranho não conseguirá se conectar ao banco de dados de uma forma diferente daquela estabelecida na criação desse mesmo usuário.

Um mesmo usuário pode ter mais de uma forma de acessar o banco de dados podendo ter, essas formas, níveis de acessos ou operações diferentes.

CREATE USER "hugo"@"213.312.231.123" **IDENTIFIED BY** "hugoprivado";
CREATE USER "hugo"@"localhost" **IDENTIFIED BY** "hugoservidor";
CREATE USER "hugo"@"%" **IDENTIFIED BY** "hugoviagem";

Dentro das [ESTRUTURAS] podemos informar o nível de ação dado a um usuário (“O quão profundo o usuário poderá ir, no seu acesso”). Sendo que cada nível hierárquico é separado por um ponto.

```
GRANT ALL ON curso_sql.* TO "hugo":"hugoservidor";  
GRANT SELECT ON curso_sql.clientes TO "hugo":"%";
```

O MySQL não permite dar permissão geral e logo depois fazer restrições específicas a essa permissão geral (dar acesso a todo o banco de dados e depois restringir o acesso a algumas tabelas, por exemplo). Sendo assim é necessário fazer permissões uma a uma, seguindo a “hierarquia”.

Tanto os comandos de ação, quanto os de estrutura são realizados a níveis de servidor (pelo MySQL Workbench), sendo assim não há a necessidade do uso do comando *use*.

Podemos fazer uma consulta para ver quais os usuários estão cadastrados e os acessos que eles possuem:

```
SELECT USER FROM mysql.user;  
SHOW GRANTS FOR "hugo"@"%";
```

8. Transações (ACID)

A DTL (Data Transaction Language) é responsável pelo conjunto de operações a serem realizadas no banco de dados, podendo ter características que garantam qualidade da transação. Sendo essas características:

ATOMICIDADE: Garante que uma operação deva ser realizada por completa, sem interferência, caso contrário, a operação é desfeita.

CONSISTÊNCIA: Garante que os dados permaneçam consistentes, mesmo que a operação falhe (retornem a um estado estável), assegurando a obediência das regras dos dados do banco.

ISOLAMENTO: Garante que cada transação realizada em um banco de dados deva ser feita de forma isolada, sem que uma interfira na outra.

DURABILIDADE: Garante que as alterações, realizadas pela/após a transação, estejam disponíveis de forma permanente, para execução das transações seguintes. Isso significa que as operações de uma transação serão definitivas se completadas com sucesso.

O MySQL possui diversos mecanismos, dentre eles existem aqueles que suportam e que não suportam transações, eles devem estar presentes na criação da tabela onde serão executadas tais operações. Para saber quais os mecanismos disponíveis, utilize o comando:

```
SHOW ENGINES;
```

Criando uma tabela e especificando o mecanismo de transação:

```
CREATE TABLE [NOME DA TABELA]
(
    [ESTRUTURA]
) ENGINE = [NOME DO MECANISMO];
```

Uma vez iniciada uma transação, a tabela onde está ocorrendo a transação é “congelada” e outras transações na mesma são deixadas em *stand by* (aguardando a transação que está ocorrendo terminar), a transação corrente só acabará quando encontrar um *commit*, fazendo a tabela adquirir os novos valores, ou um *rollback*, fazendo a tabela voltar ao estado anterior à transação corrente.

Um exemplo completo:


```

CREATE TABLE contas_bancarias
(
    id int not null auto_increment,
    funcionario varchar(50) not null,
    salario float not null,
    PRIMARY KEY (id)
) ENGINE = InnoDB;

INSERT INTO contas_bancarias(titular, saldo) VALUE ("André", 1000);
INSERT INTO contas_bancarias(titular, saldo) VALUE ("Carlos", 2000);

START TRANSACTION;

UPDATE contas_bancarias SET saldo = saldo - 100 WHERE id = 1;
UPDATE contas_bancarias SET saldo = saldo + 100 WHERE id = 2;

COMMIT;

```

9. Stored procedures e Triggers

9.1. Stored procedures

Disponíveis na maioria dos bancos de dados do mercado, os *stored procedures* (procedimentos armazenados) permitem armazenar blocos de código SQL nos mesmos, para que sejam executados em um determinado momento. Os *stored procedures* invocam ações pré-programadas quando requisitados, diferente das visões, que fazem consultas pré-programadas.

As vantagens e observação no uso da *stored procedures* são:

Centralização	Segurança	Performance/velocidade	Transações	CUIDADO!
Centraliza o código e o procedimento em questão evitando, assim, que os sistemas que utilizam o banco de dados tenham cada um seu próprio código. Sendo necessária somente a chamada no <i>Stored Procedures</i> .	Permite restringir o acesso de determinados desenvolvedores ao banco de dados, caso diferentes equipes estejam trabalhando no mesmo banco. Isso evita o acesso de pessoas não qualificadas ou mal-intencionadas.	Uma vez que os códigos são executados de forma nativa, não há necessidade de inúmeros sistemas acessarem simultaneamente, através de seus próprios códigos, informações do banco de dados, diminuindo o tráfego de informações, aumentando a performance do banco e executando o código de uma vez só no banco de dados.	Uma vez que o bloco de código está do lado do servidor, pode-se garantir a atomicidade e consistência das operações.	Dependendo da forma como as <i>Stored Procedures</i> são utilizadas, a centralização das transações pode diminuir a velocidade do banco de dados.

9.1.1. Gerenciando stored procedures

CREATE PROCEDURE [NOME]: Cria um Stored Procedure
CALL [NOME] ou **EXECUTE** [NOME]: Invoca um Stored Procedure
DROP PROCEDURE [NOME]: Excluindo uma Stored Procedure

No MySQL, como uma *stored procedures* pode usar várias linhas para sua estrutura, é necessário substituir o ; do SQL pelo \$\$ para que não de erro de interpretação - isso dentro do corpo do *stored procedure* -, no final é necessário destrocá-lo. Para isso, utiliza-se o comando *delimiter \$\$* e *delimiter ;* antes e depois do *procedure*, respectivamente.

```
DELIMITER $$  
  
CREATE PROCEDURE [NOME]()  
  
BEGIN  
[ESTRUTURA];  
  
END $$  
  
DELIMITER ;
```

OBS: No MySQL Workbench é possível criar um *procedure* clicando com o botão direito dentro em “Stored Procedures”, dentro do banco de dados corrente, e “Create Stored Procedures” e editar somente o nome e a estrutura do mesmo para a forma desejada. Depois de terminada as substituições, clique em “Apply” (isso duas vezes).

9.2. Triggers (gatilhos)

Os gatilhos, ou *triggers*, são códigos SQL executados automaticamente antes ou depois dos comandos de inserção, alteração ou exclusão de registros em tabelas no banco de dados, e até mesmo por eventos temporais (data, hora e intervalos de repetição, por exemplo). Os *triggers* possuem as mesmas

vantagens das *stored procedures*, com a diferença de que, sua execução, é baseada em eventos.

9.2.1. Gerenciando triggers

CREATE TRIGGER [NOME] [TIPO] **ON** [TABELA]: Cria um *trigger*.

DROP TRIGGER [NOME]: Exclui um *trigger*.

Os *tipos de triggers* são:

BEFORE INSERT: Executada antes de uma inserção.

BEFORE UPDATE: Executada antes de uma atualização.

BEFORE DELETE: Executada antes de uma exclusão.

AFTER INSERT: Executada depois de uma inserção.

AFTER UPDATE: Executada depois de uma atualização.

AFTER DELETE: Executada depois de uma exclusão.

TEMPORAL: Executada em eventos específicos.

Exemplo de criação de um *trigger*:

CREATE TRIGGER [NOME] [TIPO] **ON** [TABELA]

FOR EACH ROW

CALL [STORED STRUCTURE]();