Programação e Sistemas de Informação

Curso Profissional de Técnico de Gestão e Programação de Sistemas

MÓDULOS 14 E 15

Linguagem de Manipulação de Dados

Linguagem de Definição de Dados

LINGUAGEM SQL

João Martiniano

https://github.com/joaomartiniano
https://www.youtube.com/@jmartiniano

Versão 1.5.2

Conteúdo

1. Introdução

- A linguagem SQL é uma linguagem desenvolvida para trabalhar com sistemas de bases de dados relacionais
- SQL = *Structured Query Language*, ou, Linguagem de Interrogação Estruturada
- É uma linguagem declarativa, não procedimental (*nonprocedural*) ou seja, é uma linguagem que especifica o quê mas não o como
- Ou seja, através de SQL é possível especificar que se pretende um determinado tipo de dados, mas não o modo como esses dados são extraídos da base de dados
- A linguagem é composta por várias instruções que no seu conjunto permitem definir e manipular todos os aspetos de uma base de dados
- Geralmente as instruções são agrupados em três conjuntos:
    - Definição da estrutura (**DDL** ou *Data Definition Language*)
    - Manipulação de dados (**DML** ou *Data Manipulation Language*)
    - Controlo da base de dados (**DCL** ou *Data Control Language*)

História

A linguagem SQL teve a sua origem em meados dos anos 70, quando a IBM desenvolvia o protótipo de um sistema relacional chamado System R. Em paralelo, era também desenvolvida uma linguagem, a linguagem SEQUEL, para ser utilizada nesse sistema. A linguagem SEQUEL veio a dar origem à linguagem SQL (nos países anglo-saxónicos, SQL pronuncia-se "*sequel*"). Esta, após uma rápida adopção pela indústria informática, veio a ser normalizada pela ANSI e pela ISO, responsáveis pelo atual desenvolvimento da linguagem.

Conjuntos de Comandos

Os conjuntos de comandos da linguagem são os seguintes:

DDL: Definição da Estrutura	
CREATE DATABASE	Criar uma base de dados
CREATE TABLE	Criar uma tabela
ALTER TABLE	Alterar uma tabela
DROP TABLE	Eliminar uma tabela
DROP DATABASE	Eliminar uma base de dados

DML: Manipulação de Dados	
INSERT INTO	Inserir dados numa tabela
UPDATE	Atualizar os dados de uma tabela
DELETE	Apagar dados de uma tabela
SELECT	Consultar dados de uma tabela

DCL: Controlo da Base de Dados	
GRANT	Atribui permissões a utilizadores
REVOKE	Retira permissões a utilizadores

Versões

Apesar de a linguagem estar normalizada em conjunto pela ANSI e pela ISO (organismos internacionais), os fabricantes de sistemas de gestão de bases de dados normalmente oferecem uma implementação própria da linguagem.
Apesar da maioria dos comandos serem escritos e executados da mesma forma, existem, contudo, diferenças que por vezes impedem que as instruções escritas num SGBD possam ser executadas noutro. Nestes casos há que efetuar algumas adaptações.

2. Criar e Eliminar Bases de Dados

Criar uma base de dados

Para criar uma base de dados é utilizado o comando CREATE DATABASE.
A sintaxe mais básica deste comando consiste nos seguintes elementos.

Sintaxe:

```
CREATE DATABASE nome_base_dados
```

Exemplo:

```
CREATE DATABASE Clientes_Produtos;
```

Em qualquer servidor de bases são oferecidas mais opções durante a criação de uma base de dados.

Por exemplo para **MariaDB** e **MySQL**:

Sintaxe:

```
CREATE DATABASE [IF NOT EXISTS] nome_base_dados
    [CHARACTER SET charset]
    [COLLATE collation]
```

A cláusula opcional IF NOT EXISTS: se se tentar criar uma base de dados já existente, o servidor emite um erro e restantes operações (se existirem) não serão executadas. Ao utilizar esta cláusula apenas é emitido um aviso e a execução das restantes operações pode prosseguir.

As cláusulas CHARACTER SET e COLLATE permitem definir, respetivamente o *character set* (quais os caracteres que podem ser armazenados nos campos) e a *collation* (conjunto de regras que determinam como os dados textuais são ordenados e comparados).

Exemplo: Criar a base de dados loja, apenas se esta base de dados não existir

```
CREATE DATABASE IF NOT EXISTS Clientes_Produtos;
```

Exemplo: Criar a base de dados loja com um determinado *character set* e *collation*

```
CREATE DATABASE IF NOT EXISTS Clientes_Produtos
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_general_ci;
```

Eliminar uma base de dados

Para eliminar uma base de dados é utilizado o comando DROP DATABASE.

Sintaxe:

```
DROP DATABASE [IF EXISTS] nome_base_dados
```

Exemplo:

```
DROP DATABASE Clientes_Produtos;
```

A cláusula opcional IF EXISTS faz com que a base de dados apenas seja eliminada se existir. Esta funcionalidade é útil porque muito frequentemente, é necessário eliminar uma base de dados antes de a recriar mas apenas se esta já existir.

Exemplo: Eliminar (se já existir) e depois criar uma base dados

```
DROP DATABASE IF EXISTS Clientes_Produtos;

CREATE DATABASE Clientes_Produtos
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_general_ci;
```

3. Operações Sobre Tabelas

Através da DDL podemos efetuar as seguintes operações sobre tabelas:

- criar tabelas
- editar tabelas
- eliminar tabelas

Criar uma tabela

O comando CREATE TABLE permite criar uma tabela com um determinado nome, uma estrutura de campos e respetivos tipos de dados.

Sintaxe simplificada:

```
CREATE TABLE nome_tabela (
    nome_campo tipo
    ...);
```

Componentes:

`nome_tabela`	O nome da tabela
`nome_campo`	O nome do campo

Exemplo 1:

```
CREATE TABLE Clientes (
    ID          INT,
    Nome        VARCHAR(100),
    Distrito    VARCHAR(50),
    Concelho    VARCHAR(50),
    Morada      VARCHAR(255),
    Cidade      VARCHAR(100));
```

Principais tipos de dados em SQL

Cada campo necessita de ter um determinado tipo de dados. Ou seja, ao criar uma tabela, é necessário definir para cada campo qual o tipo de dados que irá conter.

Nesta área existem diferenças entre as várias implementações da linguagem SQL. Por vezes o mesmo tipo tem nomes diferentes ou então cada SGBD implementa alguns tipos de dados próprios.

Basicamente os tipos de dados dividem-se em 3 grupos:

- **Texto ou carateres**: para armazenar conjuntos de carateres (strings)
- **Numéricos**: para armazenar números; normalmente subdividem-se em:
    - Inteiros: permitem armazenar números inteiros com maior ou menor capacidade
    - Reais: permitem armazenar números reais, com maior ou menor precisão

- **Datas e horas**: armazenam datas ou horas ou uma combinação dos dois

A tabela seguinte sintetiza os principais tipos de dados:

Tipos de dados	Descrição
BOOLEAN	Valores booleanos TRUE ou FALSE
CHAR	Um único caracter
CHAR(n)	Conjunto de caracteres (string) com comprimento n
VARCHAR(n)	Conjunto variável de caracteres (de 1 a n) (se o campo for VARCHAR(10) e apenas se inserirem 5 caracteres apenas é ocupado o espaço de 5 caracteres)
TEXT	Texto até um máximo de 65.535 carateres
SMALLINT	Números inteiros de tamanho reduzido
INT ou INTEGER	Números inteiros de tamanho médio
NUMERIC NUMERIC (m,d) DECIMAL DECIMAL (m,d)	Utilizado para armazenar números com representação exata. m = número total de dígitos armazenados n = número de dígitos após o ponto decimal (exemplo: `Numeric(5,2)` → `123,45`)
FLOAT REAL DOUBLE	Números de vírgula flutuante (*floating point*). Utilizado para armazenar números que não têm uma representação exata.
DATE	Data no formato 'Ano-Mês-Dia' (exemplo: '2022-01-26')
TIME	Hora no formato 'HH:MM:SS' (exemplo: '15:01:56')
DATETIME	Combinação de data e hora no formato Ano-Mês-Dia HH:MM:SS (exemplo: '2022-01-26 15:01:56')

Acerca do tipo BOOLEAN

Os SGBD MySQL e MariaDB não implementam diretamente o tipo BOOLEAN.

Este tipo de dados é emulado utilizando o tipo TINY INT (este tipo armazena valores numéricos inteiros). Neste caso o valor 0 assume o significado de FALSE (falso) e qualquer valor diferente de 0 assume o valor TRUE (verdadeiro).

Deste modo, aquando da criação de uma tabela, qualquer campo especificado com o tipo BOOLEAN é automaticamente convertido para TINYINT.
De igual modo podem ser utilizados os valores TRUE e FALSE em instruções, sendo os mesmos automaticamente convertidos para 1 e 0, respetivamente.

Definição complexa

A sintaxe apresentada anteriormente para a criação de uma tabela é uma sintaxe simplificada. A seguinte sintaxe apresenta mais alguns elementos, essenciais para a criação de uma tabela.

Sintaxe:

```
CREATE TABLE nome_tabela (
    nome_campo tipo [NOT NULL] [DEFAULT valor] [AUTO_INCREMENT] [PRIMARY KEY],
    ...

    [FOREIGN KEY (chave_estrangeira) REFERENCES
         tabela_estrangeira (campo)]
);
```

Componentes:

nome_tabela	O nome da tabela
nome_campo	O nome do campo
NOT NULL	O campo não pode conter valores nulos (NULL)
DEFAULT	Permite especificar um valor por defeito para o campo
AUTO_INCREMENT	É automaticamente atribuído um valor numérico ao campo. No registo seguinte este valor é incrementado em 1
PRIMARY KEY	O campo é a chave primária ou faz parte da chave primária
FOREIGN KEY	Permite criar uma chave estrangeira

Exemplo:

```
CREATE TABLE Clientes (
    ID              INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    Nome            VARCHAR(100) NOT NULL,
    Distrito        VARCHAR(50),
    Concelho        VARCHAR(50),
    Morada          VARCHAR(255),
    Cidade          VARCHAR(100));

CREATE TABLE Produtos (
    ID              INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    Tipo            VARCHAR(50) NOT NULL,
    Designacao      VARCHAR(100) NOT NULL,
    Preco           NUMERIC(10,2) NOT NULL DEFAULT 0.0,
    Cor             VARCHAR(20),
    Descontinuado   BOOL);
```

Chave estrangeira

A definição de chaves estrangeiras é feita utilizando a cláusula FOREIGN KEY:

```
FOREIGN KEY (chave_estrangeira) REFERENCES
    tabela_estrangeira (campo)
```

Em que:

chave_estrangeira	O campo da chave estrangeira
tabela_estrangeira	A outra tabela à qual a tabela que está a ser criada deverá ficar relacionada
campo	O nome do campo da outra tabela, relacionado com a chave estrangeira

Exemplo
Imaginemos duas tabelas, Proprietarios e Casas, em que um proprietário pode ser proprietário de uma casa:

```
Casas(ID, Endereco, Cor, Idade, Preco)
Proprietarios(ID, Nome, Telefone, Casa_ID)
```

Neste caso, o campo Casa_ID da tabela Proprietarios é chave estrangeira, que implementa o relacionamento entre um proprietário e a respetiva casa.

O código SQL para a criação das tabelas seria:

```
CREATE TABLE Casas (
    ID          INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    Endereco    VARCHAR(255) NOT NULL,
    Cor         VARCHAR(15) NOT NULL,
    Idade       INT,
    Preco       NUMERIC(10,2));

CREATE TABLE Proprietarios (
    ID          INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    Nome        VARCHAR(100) NOT NULL,
    Telefone    VARCHAR(9) NOT NULL,
    Casa_ID     INT,
    FOREIGN KEY (Casa_ID) REFERENCES Casas (ID));
```

Eliminar uma tabela

O comando DROP TABLE permite eliminar uma tabela.

Sintaxe:

```
DROP TABLE nome_tabela
```

Exemplo:

```
DROP TABLE Clientes;
```

Alterar a estrutura de uma tabela

O comando ALTER TABLE permite alterar a estrutura de uma tabela, nomeadamente:

- adicionar campos
- alterar campos
- eliminar campos

Adicionar campos

Podemos adicionar um novo campo a uma tabela com a seguinte instrução:

Sintaxe:

```
ALTER TABLE nome_tabela
ADD campo tipo
```

Exemplo:

```
ALTER TABLE Produtos
ADD Quantidade INT;
```

Alterar campos

Podemos modificar o tipo de dados de um campo com a seguinte instrução:

Sintaxe:

```
ALTER TABLE nome_tabela
MODIFY campo tipo
```

Exemplo:

```
ALTER TABLE Produtos
MODIFY Quantidade NUMERIC(10,2);
```

Eliminar campos

Podemos remover um campo de uma tabela com a seguinte instrução:

Sintaxe:

```
ALTER TABLE nome_tabela
DROP campo
```

Exemplo:

```
ALTER TABLE Produtos
DROP Quantidade;
```

Índices

Um índice é um mecanismo para acelerar consultas aos dados de uma tabela, podendo ser composto por um ou mais campos.

Importa saber que a utilização de índices, embora aumente a velocidade das operações de consulta, é um mecanismo que, por outro lado, poderá ter um impacto negativo na performance do sistema em certas situações. Isto porque obriga o SGBD a manter sempre atualizados, dados associados aos índices.

Adicionar um índice

Podemos adicionar um índice a uma tabela, utilizando a seguinte instrução:

```
CREATE [UNIQUE] INDEX nome_índice
ON tabela (campo1 [, campo2 ...])
```

Em que:

`UNIQUE`	(opcional) Obriga a que o valor do campo não possa conter valores repetidos
`nome_índice`	O nome do índice
`tabela`	A tabela na qual é criado o índice
`campo`	O campo ou campos que compõem o índice

Exemplo 1: Criar um índice na tabela Produtos, para o campo Preco

```
CREATE INDEX idx_Preco
ON Produtos (Preco);
```

Exemplo 2: Criar um índice na tabela Clientes, composto pelos campos Distrito e Concelho

```
CREATE INDEX idx_DistConc
ON Clientes (Distrito, Concelho);
```

Eliminar um índice

A instrução DROP INDEX permite eliminar um índice:

Sintaxe:

```
DROP INDEX nome_índice ON tabela
```

Exemplo:

```
DROP INDEX idx_Preco ON Produtos;
```

4. Operações sobre dados

Outra funcionalidade oferecida pela linguagem SQL são os comandos de manipulação de dados, os quais fazem parte da DML (*Data Manipulation Language*).

Através dos comandos da DML é possível efetuar as seguintes operações:

- inserir dados
- alterar dados
- eliminar dados
- consultar dados

Inserir dados em tabelas

O comando INSERT INTO permite inserir dados, na forma de registos, na base de dados.

Sintaxe:

```
INSERT INTO tabela (campo1, campo2, ... )
VALUES (valor1, valor2, ...)
```

Exemplo 1:

```
INSERT INTO Clientes (Nome, Distrito, Concelho, Morada, Cidade)
VALUES ('Margarida Gomes', 'Porto', 'Porto', 'Rua Afonso
Baldaia', 'Porto');
```

Exemplo 2:

```
INSERT INTO Produtos (Designacao, Preco, Cor)
VALUES ('Microsoft Office 2007', 190, NULL), ('Computador
Portátil Asus X200', 722.3, 'Cinza');
```

Note-se que no exemplo anterior foram inseridos dois registos com apenas uma instrução INSERT.

Alterar dados

O comando UPDATE permite alterar os valores de campos num determinado conjunto de registos.

Sintaxe (simplificada):

```
UPDATE tabela
SET campo1 = valor, campo2 = valor, ...
[WHERE condição]
```

Exemplo 1: Aumentar o preço de todos os produtos em 5 cêntimos

```
UPDATE Produtos
SET Preco = Preco + 0.05;
```

A cláusula WHERE
A cláusula WHERE permite modificar apenas certos registos. Ou seja, apenas serão modificados os registos que respeitem uma determinada condição (especificada nesta cláusula).

Exemplo 2: Aumentar em 5 cêntimos o preço de todos os produtos cujo preço é inferior ou igual a 10

```
UPDATE Produtos
SET Preco = Preco + 0.05
WHERE Preco <= 10;
```

Eliminar dados

O comando DELETE permite eliminar dados das tabelas, nomeadamente um ou mais registos.

Sintaxe:

```
DELETE FROM tabela
[WHERE condição]
```

Exemplo 1: Apagar todos os registos da tabela Clientes

```
DELETE FROM Clientes;
```

A cláusula WHERE
Tal como no comando UPDATE, a cláusula WHERE permite apenas eliminar certos registos.

Exemplo 2: Apagar todos os produtos cujo preço seja superior a 100

```
DELETE FROM Produtos
WHERE Preco > 100;
```

Exemplo 3: Apagar todos os registos dos clientes cujos nomes não começam com a letra 'A'

```
DELETE FROM Clientes
WHERE NOT(Nome LIKE 'A%');
```

5. Consultar Dados

Definida a estrutura de uma base de dados e após a inserção de dados na mesma, é possível passar a extrair dados. De facto, esta é a principal razão para a utilização de uma base de dados: não só a possibilidade de armazenar informação, mas principalmente, a capacidade de extrair dados, relacioná-los entre si e eventualmente transformá-los, acrescentado valor à informação.

Na linguagem SQL é utilizada a instrução SELECT para efetuar consultas aos dados. Para começar far-se-á uma primeira abordagem simplificada.

Sintaxe simplificada:

```
SELECT campos
FROM tabela(s)
[WHERE condição]
```

Componentes da instrução SELECT:

`campos`	O campo ou campos cujos valores interessa consultar
`tabela(s)`	A tabela ou tabelas nas quais são pesquisados os campos
`condição`	Condição opcional que permite especificar a seleção de determinados registos (ou seja, apenas são devolvidos dados que respeitem a condição)

Exemplo 1: Obter o nome e o distrito de todos os clientes

```
SELECT Nome, Distrito
FROM Clientes;
```

O operador *

Muitas vezes é necessário obter todos os campos de uma tabela. Nestas situações para poupar tempo, em vez de escrever a designação de todos os campos, é utilizado o operador *, o qual significa "obter todos os campos de uma tabela".

Exemplo 2: Obter todos os campos da tabela Clientes

```
SELECT *
FROM Clientes;
```

A consulta anterior é equivalente a esta:

```
SELECT ID, Nome, Distrito, Concelho, Morada, Cidade
FROM Clientes;
```

Condições

Nos exemplos anteriores as consultas retornam todos os registos existentes na tabela Clientes. Normalmente, apenas interessa obter um determinado conjunto de registos, que respondam a uma determinada condição.

Exemplo 3: Obter a designação e preço de todos os produtos cujo preço é superior a 100

```
SELECT Designacao, Preco
FROM Produtos
WHERE Preco > 100;
```

Operadores

A linguagem SQL possui operadores que quando utilizados na cláusula WHERE permitem efetuar condições que respondem a um grande conjunto de situações.

Operadores de Comparação

Operador	Descrição
=	Igual (*equal to*)
<>	Diferente (*not equal to*)
<	Menor (*less than*)
>	Maior (*greater than*)
<=	Menor ou igual (*less than or equal to*)
>=	Maior ou igual (*greater than or equal to*)

Exemplo 4: Obter a designação e preço de todos os produtos cujo preço é inferior ou igual a 10

```
SELECT Designacao, Preco
FROM Produtos
WHERE Preco <= 10;
```

Exemplo 5: Obter o nome de todos os clientes que não residem em Lisboa

```
SELECT Nome
FROM Clientes
WHERE Cidade <> 'Lisboa';
```

Operadores Lógicos

Operador	Descrição
AND	**E** (conjunção) Junta duas ou mais condições, e faz com que só sejam mostrados os registos que satisfaçam todas as condições especificadas
OR	**OU** (disjunção) Junta duas ou mais condições, e faz com que sejam mostrados os registos que satisfaçam qualquer uma das condições especificadas
NOT	**Não** (negação) Só são mostrados os registos que **não** respeitem uma determinada condição

Exemplo 6: Obter a designação e preço de todos os produtos cujo preço está compreendido entre 200 (inclusive) e 400 (inclusive)

```
SELECT Designacao, Preco
FROM Produtos
WHERE Preco >= 200 AND Preco <= 400;
```

Exemplo 7: Obter a designação e preço de todos os produtos cujo preço está compreendido entre 200 (inclusive) e 400 (inclusive), e cuja cor é verde

```
SELECT Designacao, Preco
FROM Produtos
WHERE Preco >= 200 AND Preco <= 400 AND Cor = 'Verde';
```

Exemplo 8: Obter a designação e preço de todos os produtos cujo preço é igual a 200 ou cuja cor é azul

```
SELECT Designacao, Preco
FROM Produtos
WHERE Preco = 200 OR Cor = 'Azul';
```

Exemplo 9: Obter a designação e preço de todos os produtos cujo preço não é igual a 200 e cuja cor não é azul

```
SELECT Designacao, Preco
FROM Produtos
WHERE NOT(Preco = 200 OR Cor = 'Azul');
```

Note-se que a consulta anterior também poderia ser escrita da seguinte forma:

```
SELECT Designacao, Preco
FROM Produtos
WHERE Preco <> 200 AND Cor <> 'Azul';
```

Outros Operadores de Comparação

Operador	Descrição
BETWEEN	**Compreendido entre...** Testa se um valor está dentro de um determinado intervalo
IN	**Dentro de...** Testa se o valor de uma expressão é igual a um dos valores de uma lista
LIKE	**Semelhante a...** Testa se uma string é semelhante a outra

Exemplo 10: Obter a designação e preço de todos os produtos cujo preço está compreendido entre 200 (inclusive) e 400 (inclusive)

```
SELECT Designacao, Preco
FROM Produtos
WHERE Preco BETWEEN 200 AND 400;
```

Exemplo 11: Obter a designação, preço e cor dos produtos cuja cor é azul, vermelho ou cinza

```
SELECT Designacao, Preco, Cor
FROM Produtos
WHERE Cor IN ('Azul', 'Vermelho', 'Cinza');
```

Note-se que a consulta anterior também poderia ser escrita da seguinte forma:

```
SELECT Designacao, Preco, Cor
FROM Produtos
WHERE Cor = 'Azul' OR Cor = 'Vermelho' OR Cor = 'Cinza';
```

O operador LIKE

O operador LIKE permite testar se uma string é semelhante a outra. Para tal devem-se utilizar os carateres % e ? os quais têm o seguinte significado:

%	Qualquer conjunto de carateres
?	Qualquer carater

Exemplo 12: Obter a designação de todos os produtos que começam com o carater P

```
SELECT Designacao
FROM Produtos
WHERE Designacao LIKE 'P%';
```

Exemplo 13: Obter os produtos em cuja designação está a palavra 'Office'

```
SELECT *
FROM Produtos
WHERE Designacao LIKE '%Office%';
```

Ordenação de Resultados

Uma funcionalidade bastante útil da linguagem SQL é a possibilidade de ordenar os resultados de uma consulta. A ordenação é efetuada através da cláusula ORDER BY e permite:

- ordenar de acordo com um ou mais campos
- especificar o tipo de ordenação (ascendente ou descendente)

A sintaxe da instrução SELECT, utilizando a cláusula ORDER BY passa a ser:

```
SELECT campos
FROM tabela(s)
[WHERE condição]
[ORDER BY <campo1> [ASC|DESC] [,<campo2> [ASC|DESC] [,...]]]
```

De notar que a cláusula ORDER BY é opcional, deve aparecer após a cláusula WHERE e tem os seguintes componentes:

campo1, ...	O campo ou campos que ordenam os resultados
ASC|DESC	Ascendente ou descendente: o tipo de ordenação; cada campo pode ter um (e só um) tipo de ordenação (por defeito a ordenação de qualquer campo é ascendente)

Exemplo 14: Obter todos os produtos, ordenados de forma crescente pelo preço

```
SELECT *
FROM Produtos
ORDER BY Preco;
```

Exemplo 15: Obter todos os produtos de cor vermelha, ordenados de forma decrescente pelo preço

```
SELECT *
FROM Produtos
WHERE Cor = 'Vermelho'
ORDER BY Preco DESC;
```

Exemplo 16: Obter dados de todos os clientes, ordenados em primeiro lugar pelo distrito e depois pelo concelho

```
SELECT *
FROM Clientes
ORDER BY Distrito, Concelho;
```

Os predicados ALL e DISTINCT

O predicado DISTINCT elimina dados duplicados do resultado de uma consulta.

Sintaxe:

```
SELECT [ALL | DISTINCT] campos
FROM tabela(s)
[WHERE condição]
```

Exemplo: Mostrar todos os distritos registados na tabela Clientes

```
SELECT Distrito
FROM Clientes;
```

O resultado desta consulta contém dados duplicados:

Distrito
Porto
Faro
Porto
Coimbra
Coimbra
Braga
Porto

Exemplo: Mostrar todos os distritos registados na tabela Clientes, sem dados duplicados

```
SELECT DISTINCT Distrito
FROM Clientes;
```

Resultado:

Distrito
Porto
Faro
Coimbra
Braga

O predicado ALL retorna todos os dados. É de utilização redundante porque, por defeito, são devolvidos todos os registos. Assim, as consultas seguintes são equivalentes:

```
SELECT Distrito
FROM Clientes;
```

⇔

```
SELECT ALL Distrito
FROM Clientes;
```

6. Funções de Agregação

A linguagem SQL disponibiliza cinco funções denominadas funções de agregação, isto é, funções que resumem o resultado de uma consulta, ao invés de mostrarem todos os registos retornados. Utilizando estas funções obtém-se apenas um valor e não um conjunto de registos.

Existe 5 funções de agregação:

`COUNT()`	Efetua contagens de registos
`SUM()`	Efetua soma de valores
`AVG()`	Calcula a média de valores
`MAX()`	Devolve o valor máximo de um conjunto de valores
`MIN()`	Devolve o valor mínimo de um conjunto de valores

Função COUNT()

A função COUNT() conta o número de registos resultantes de uma consulta.

Sintaxe:

```
SELECT COUNT(campo)
FROM tabela(s)
[WHERE condição]
```

Se não for utilizado o carater asterisco (*), a função não conta com os registos cujo valor do campo especificado seja NULL. Se o asterisco for especificado é calculado o número total de registos, com ou sem valores nulos (a utilização do asterisco torna a consulta mais rápida).

Exemplo 17: Contar o número de produtos existentes na tabela

```
SELECT COUNT(*)
FROM Produtos;
```

Exemplo 18: Contar o número de produtos existentes na tabela, cuja cor é azul

```
SELECT COUNT(*)
FROM Produtos
WHERE Cor = 'Azul';
```

Função SUM()

A função SUM() calcula a soma dos valores de um dado campo.

Sintaxe:

```
SELECT SUM(campo)
FROM tabela(s)
[WHERE condição]
```

Exemplo 19: Obter a soma do preço de todos os produtos cujo preço é superior a 100

```
SELECT SUM(Preco)
FROM Produtos
WHERE Preco > 100;
```

Função AVG()

A função AVG() calcula o valor médio de um dado campo.

Sintaxe:

```
SELECT AVG(campo)
FROM tabela(s)
[WHERE condição]
```

Exemplo 20: Obter o preço médio de todos os computadores portáteis

```
SELECT AVG(Preco)
FROM Produtos
WHERE Tipo = 'Computador Portátil';
```

Função MAX()

A função MAX() retorna o maior valor de um campo.

Sintaxe:

```
SELECT MAX(campo)
FROM tabela(s)
[WHERE condição]
```

Exemplo 21: Obter o maior preço de produto

```
SELECT MAX(Preco)
FROM Produtos;
```

Função MIN()

A função MIN() retorna o menor valor de um campo.

Sintaxe:

```
SELECT MIN(campo)
FROM tabela(s)
[WHERE condição]
```

Exemplo 22: Obter o preço mais baixo dos computadores portáteis

```
SELECT MIN(Preco)
FROM Produtos
WHERE Tipo = 'Computador Portátil';
```

7. Agrupamento de Dados

À medida que as tabelas vão crescendo com dados, surge a necessidade de obter dados agrupados segundo um determinado critério. Neste caso utilizam-se as instruções GROUP BY e HAVING. Normalmente os dados são obtidos em conjugação com as funções de agregação (SUM, COUNT, etc.).

GROUP BY

A instrução GROUP BY permite agregar os dados:

- numa primeira fase a instrução SELECT obtém vários registos (estes registos podem ser filtrados por uma cláusula WHERE)
- de seguida, a instrução GROUP BY agrupa os resultados de acordo com um ou mais campos

Sintaxe:

```
SELECT campo(s)
FROM tabela
[WHERE condição]
[GROUP BY campo(s)]
```

É importante notar que os campos especificados na instrução SELECT deverão estar também presentes na cláusula GROUP BY. O contrário, contudo, não é verdade: podem existir campos na cláusula GROUP BY que não estão presentes na instrução SELECT.

Exemplo 1: Apresentar para cada cor, o número de produtos existentes

```
SELECT Cor, COUNT(*)
FROM Produtos
GROUP BY Cor;
```

Exemplo 2: Ordenar o exemplo anterior, de forma crescente, pelo resultado da contagem

```
SELECT Cor, COUNT(*) AS Contagem
FROM Produtos
GROUP BY Cor
ORDER BY Contagem;
```

Exemplo 3: Obter o número de clientes de cada cidade

```
SELECT Cidade, Count(*)
FROM Clientes
GROUP BY Cidade;
```

Exemplo 4: Obter o preço mais elevado para cada tipo de produto

```
SELECT Tipo, MAX(Preco)
FROM Produtos
GROUP BY Tipo;
```

HAVING

A cláusula HAVING é utilizada em conjunto com a GROUP BY e permite restringir os grupos de resultados.
Apesar de ser similar à cláusula WHERE, ambas têm finalidades diferentes:

- a cláusula WHERE filtra registos individuais
- a cláusula HAVING filtra grupos de registos

Sintaxe:

```
SELECT campo(s)
FROM tabela
[WHERE condição]
[GROUP BY campo(s)]
[HAVING condição]
```

Exemplo 1: Obter o número de clientes de cada cidade, no caso em que as cidades têm mais do que um cliente

```
SELECT Cidade, Count(*)
FROM Clientes
GROUP BY Cidade
HAVING Count(*) > 1;
```

8. Consultas envolvendo várias tabelas

Nos exemplos analisados até aqui, apenas se obtiveram dados de uma tabela. No entanto, o poder das bases de dados relacionais, está, como o nome indica, na capacidade de relacionar dados.
Ou seja, apresentar dados de tabelas distintas e que estão relacionados.

Produto cartesiano

No caso mais simples podem-se simplesmente selecionar campos de duas tabelas.

Exemplo 1: Mostrar todos os campos das tabelas Clientes e Produtos

```
SELECT *
FROM Clientes, Produtos;
```

Exemplo 2: Mostrar apenas um campo de ambas as tabelas

```
SELECT Clientes.Nome, Produtos.Designacao
FROM Clientes, Produtos;
```

Alias

Uma característica interessante da linguagem SQL e que é bastante útil quando se desenvolvem consultas envolvendo várias tabelas e vários campos, consiste na utilização de *alias*.
Basicamente um *alias* é um substituto para o nome de uma tabela ou campo. Normalmente utilizam-se *aliases* em duas situações:

- mudar o nome de um campo para que no resultado de uma consulta apareça com uma designação mais apropriada
- substituir o nome de uma tabela, muitas vezes para uma abreviatura, com o intuito de poupar espaço e tornar o código mais legível

Utilizando *aliases*, o exemplo anterior pode ser reescrito da seguinte forma:

```
SELECT C.Nome AS 'Nome do Cliente', P.Designacao AS 'Designação
do Produto'
FROM Clientes AS C, Produtos AS P;
```

Junções (*joins*)

As consultas anteriores não fazem grande sentido uma vez que o resultado é apenas a combinação de todos os registos de ambas as tabelas. O que faz sentido no entanto, é efetuar restrições, de modo a mostrar apenas os dados que estão relacionados.
Para tal, é necessário criar uma nova tabela, a tabela Encomendas, que irá permitir relacionar as tabelas Clientes e Produtos.

```
CREATE TABLE Encomendas (
  ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  Cliente_ID INT NOT NULL,
  Produto_ID INT NOT NULL,
  Quantidade INT);
```

Nesta tabela são armazenadas as encomendas que os clientes efetuam de produtos, e as respetivas quantidades.

Exemplo das tabelas e alguns dados:

Clientes

ID	Nome	Distrito	Concelho	Morada	Cidade
1	Margarida Gomes	Porto	Porto	Rua Afonso Baldaia	Porto
2	Mário Cordeiro	Braga	Guimarães	Avenida D. João IV	Guimarães

Produtos

ID	Tipo	Designação	Preço	Cor
1	Software	Microsoft Office 2007	190	NULL
2	Software	Microsoft Office 2010	200.55	NULL
3	Computador Portátil	Asus X200	722.3	Cinza

Encomendas

ID	Cliente_ID	Produto_ID	Quantidade
1	2	1	15
2	2	2	20
3	1	1	1
4	1	2	NULL

À operação de juntar registos chama-se junção ou *join*. Existem vários tipos de *joins*, sendo que os principais são:

- *Equijoin*
- *Inner join*
- *Left outer join*
- *Right outer join*

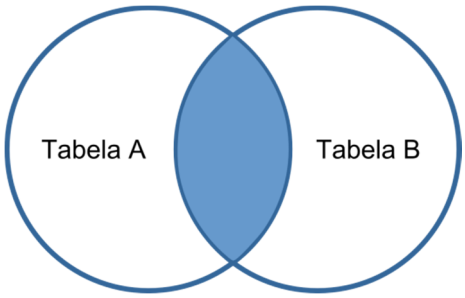
Equijoin

Uma *equijoin* é o tipo de junção mais simples e intuitiva de efetuar: a ligação entre tabelas é feita através da igualdade de campos, das várias tabelas que se pretendem juntar.

Sintaxe:

```
SELECT campo(s)
FROM TabelaA, TabelaB
WHERE TabelaA.campo = TabelaB.campo
```

Uma *equijoin* retorna registos quando existe correspondência entre a tabela A e a tabela B. Em termos visuais o resultado é o seguinte:



Exemplo 1: Saber quais os clientes que efetuaram encomendas (mostrar o ID, o nome e a cidade)

```
SELECT Clientes.ID, Clientes.Nome, Clientes.Cidade
FROM Clientes, Encomendas
WHERE Clientes.ID = Encomendas.Cliente_ID;
```

Neste caso os registos dos clientes aparecem múltiplas vezes, porque correspondem ao número de encomendas que efetuaram. Para que não ocorram repetições deve-se utilizar o predicado DISTINCT:

```
SELECT DISTINCT Clientes.ID, Clientes.Nome, Clientes.Cidade
FROM Clientes, Encomendas
WHERE Clientes.ID = Encomendas.Cliente_ID;
```

Exemplo 2: Para cada encomenda, mostrar o ID da encomenda, o nome do cliente e a quantidade encomendada

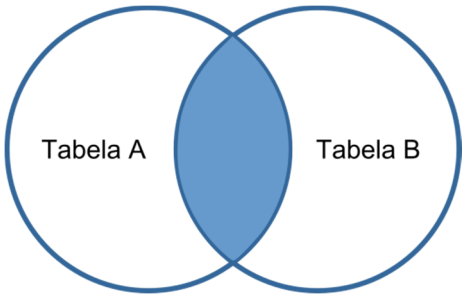
```
SELECT E.ID, C.Nome AS 'Nome Cliente', E.Quantidade AS
'Quantidade Encomendada'
FROM Clientes AS C, Encomendas AS E
WHERE C.ID = E.Cliente_ID;
```

Exemplo 3: Para cada encomenda, mostrar o ID da encomenda, o nome do cliente, a designação do produto e a quantidade encomendada

```
SELECT E.ID, C.Nome AS 'Nome Cliente', P.Designacao AS 'Produto
Encomendado', E.Quantidade AS 'Quantidade Encomendada'
FROM Clientes AS C, Encomendas AS E, Produtos as P
WHERE C.ID = E.Cliente_ID AND P.ID = E.Produto_ID;
```

Inner Join

As consultas que são efetuadas através de *equijoin* podem também ser efetuadas através de uma *inner join*, sendo os resultados exatamente iguais.



Sintaxe:

```
SELECT campo(s)
FROM TabelaA
INNER JOIN TabelaB ON TabelaA.campo = TabelaB.campo
```

Os exemplos anteriores podem ser modificados de modo a ser utilizada uma *inner join*.

Exemplo 1: Determinar quais os clientes que efetuaram encomendas (mostrar o ID, o nome e a cidade)

```
SELECT DISTINCT Clientes.ID, Clientes.Nome, Clientes.Cidade
FROM Clientes
INNER JOIN Encomendas ON Clientes.ID = Encomendas.Cliente_ID;
```

Exemplo 2: Para cada encomenda, mostrar o ID da encomenda, o nome do cliente e a quantidade encomendada

```
SELECT E.ID, C.Nome AS 'Nome Cliente', E.Quantidade AS
'Quantidade Encomendada'
FROM Clientes AS C
INNER JOIN Encomendas AS E ON C.ID = E.Cliente_ID;
```

Exemplo 4: Para cada encomenda, mostrar o ID da encomenda, o nome do cliente, a designação do produto e a quantidade encomendada

```
SELECT E.ID, C.Nome AS 'Nome Cliente', P.Designacao AS 'Produto
Encomendado', E.Quantidade AS 'Quantidade Encomendada'
FROM Clientes AS C
INNER JOIN Encomendas AS E ON C.ID = E.Cliente_ID
INNER JOIN Produtos AS P ON E.Produto_ID = P.ID;
```

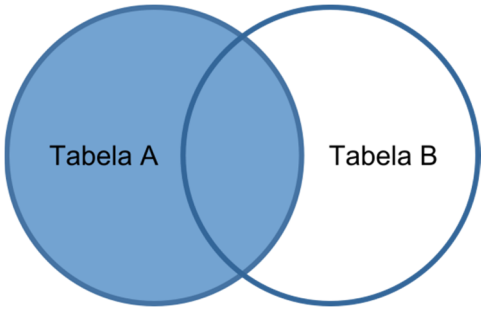
Utilização da cláusula WHERE
Tal como numa consulta normal, é possível utilizar a cláusula WHERE para restringir os resultados.

Pegando na consulta do exemplo 2, esta é modificada de modo a mostrar apenas os dados das encomendas cuja quantidade é superior a 15:

```
SELECT E.ID, C.Nome AS 'Nome Cliente', E.Quantidade AS
'Quantidade Encomendada'
FROM Clientes AS C
INNER JOIN Encomendas AS E ON C.ID = E.Cliente_ID
WHERE E.Quantidade > 15;
```

Left Outer Join

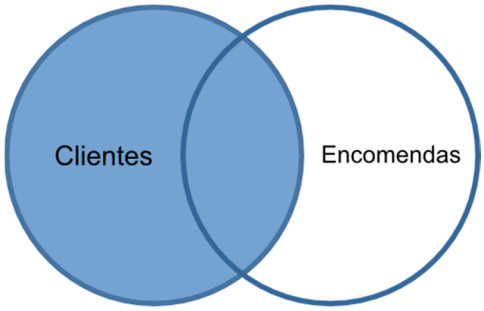
Com uma *left outer join* é possível obter todos os registos da tabela A bem como os registos correspondentes na tabela B.



Sintaxe:

```
SELECT campo(s)
FROM TabelaA LEFT JOIN TabelaB
ON TabelaA.campo = TabelaB.campo
```

Exemplo 1: Obter dados de <u>todos</u> os clientes bem como das encomendas que efetuaram



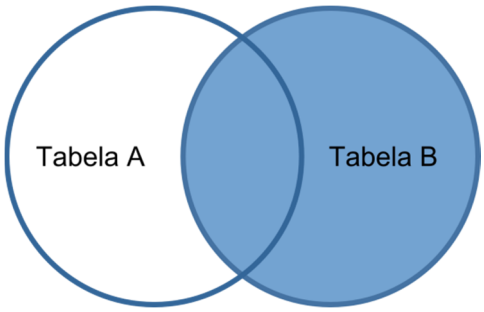

```
SELECT C.Nome AS 'Nome Cliente', C.Cidade, E.Produto_ID,
E.Quantidade
FROM Clientes AS C
LEFT JOIN Encomendas AS E ON C.ID = E.Cliente_ID;
```

Exemplo 2: Obter dados de <u>todos</u> os clientes de Coimbra, bem como das encomendas que efetuaram

```
SELECT C.Nome AS 'Nome Cliente', C.Cidade, E.Produto_ID,
E.Quantidade
FROM Clientes AS C
LEFT JOIN Encomendas AS E ON C.ID = E.Cliente_ID
WHERE C.Cidade = 'Coimbra';
```

Right Outer Join

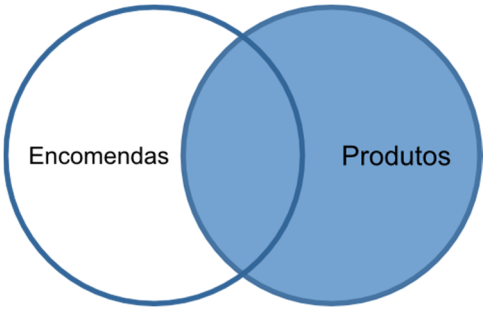
Através de uma *right outer join* é possível obter todos os registos da tabela B bem como os registos correspondentes na tabela A.



Sintaxe:

```
SELECT campo(s)
FROM TabelaA RIGHT JOIN TabelaB
ON TabelaA.campo = TabelaB.campo
```

Exemplo 1: Obter dados de <u>todos</u> os produtos bem como das respetivas encomendas




```
SELECT P.Designacao AS 'Designação Produto', P.Preco AS
'Preço', P.Cor, E.Cliente_ID, E.Quantidade
FROM Produtos AS P
RIGHT JOIN Encomendas AS E ON P.ID = E.Produto_ID;
```

Exemplo 2: Obter dados de todos os produtos do tipo *software* bem como das respetivas encomendas

```
SELECT P.Designacao AS 'Designação Produto', P.Preco AS
'Preço', P.Cor, E.Cliente_ID, E.Quantidade
FROM Produtos AS P
RIGHT JOIN Encomendas AS E ON P.ID = E.Produto_ID
WHERE P.Tipo = 'Software';
```

Subqueries

Uma *subquery* (ou subconsulta), não é mais do que uma consulta dentro de outra consulta. Na maior parte das vezes, uma subconsulta pode ser reescrita e implementada através de uma *join*. Regra geral devem-se evitar *subqueries*, por tornarem mais lenta a execução das consultas.

Existem diferentes formas *subqueries*, sendo apresentada de seguida a mais habitual, isto é, uma *subquery* na cláusula WHERE.

Sintaxe:

```
SELECT campo(s)
FROM tabela
WHERE condição IN (SELECT campo(s)
                   FROM tabela
                   [WHERE condição]);
```

Assim sendo, temos duas consultas:

- a primeira, a subconsulta, retorna um valor ou um conjunto de valores (uma tabela) de acordo com um determinado critério
- a outra consulta, a consulta principal, vai atuar sobre os dados da subconsulta e filtrá-los de acordo com um determinado critério

Exemplo: Obter o nome dos clientes que efetuaram encomendas

```
SELECT Nome
FROM Clientes
WHERE ID IN (SELECT Cliente_ID
             FROM Encomendas);
```

Na consulta anterior:

- a subquery SELECT Client_ID FROM Encomendas, retorna a chave primária de todos os clientes com registo na tabela Encomendas (ou seja, todos os clientes que efetuaram encomendas)
- com base nesses dados, é mostrado o nome desses clientes

Uniões (*unions*)

Uma união (ou *union*) permite juntar dois ou mais comandos SELECT e obedece às seguintes restrições:

- o número de campos selecionados tem, obrigatoriamente, de ser igual
- os tipos de dados dos campos correspondentes nas diferentes consultas, deverão ser iguais
- apenas poderá existir uma única cláusula ORDER BY, a qual deve ser colocada no último SELECT

As informações dos vários SELECT não têm necessariamente de estar relacionadas, embora o número de campos selecionados tenha de ser igual. Além disso, apenas poderá existir uma

Sintaxe:

```
SELECT campos
FROM tabela(s)
[WHERE condição]
UNION
SELECT campos
FROM tabela(s)
[WHERE condição]
[UNION
...
]
[ORDER BY campo(s)]
```

Exemplo

Imaginando que existem duas tabelas, Carros e Motos, com a seguinte estrutura:

Carros

ID	Matricula	Marca	Modelo	Cilindrada	NumeroPortas
1	00-AA-11	Audi	A3	1600	3
2	21-NC-14	Citroen	C3	1400	5

Motos

ID	Matrícula	Marca	Modelo	Cilindrada
1	79-NA-79	Honda	Goldwing	1500
2	80-AV-80	Yamaha	FJR	1300
3	01-OP-45	Kawasaki	Z 750	750

A seguinte consulta retorna dados de ambas as tabelas:

```
SELECT Matricula, Marca, Modelo
FROM Carros
UNION
SELECT Matricula, Marca, Modelo
FROM Motos;
```

Resultado:

Matricula	Marca	Modelo
00-AA-11	Audi	A3
21-NC-14	Citroen	C3
79-NA-79	Honda	Goldwing
80-AV-80	Yamaha	FJR
01-OP-45	Kawasaki	Z 750

Dados da tabela Carros

Dados da tabela Motos

Por outro lado, a seguinte consulta irá dar origem a um erro porque o número de campos selecionados no primeiro SELECT é diferente relativamente ao segundo SELECT:

```
SELECT Matricula, Marca, Modelo, NumeroPortas
FROM Carros
UNION
SELECT Matricula, Marca, Modelo
FROM Motos;
```

A seguinte consulta também teria como resultado um erro porque os campos selecionados correspondentes são de tipos diferentes (ID no primeiro SELECT e Matricula no segundo SELECT):

```
SELECT ID, Marca
FROM Carros
UNION
SELECT Matricula, Marca
FROM Motos;
```

Para finalizar, tal como nas consultas normais, é possível utilizar a cláusula WHERE de modo a restringir os dados obtidos:

```
SELECT Matricula, Marca, Modelo
FROM Carros
WHERE Marca = 'Nissan'
UNION
SELECT Matricula, Marca, Modelo
FROM Motos
WHERE Cilindrada < 600;
```

9. Transações

O conceito de transação é muito importante num SGBD, porque este é o único responsável pela validade dos dados armazenados.

Por exemplo, imaginemos uma transferência entre duas contas bancárias **A** e **B**, através da qual se pretende movimentar a quantia de 5.000€ de **A** para **B**. A transferência é composta por duas operações:

1ª Operação: debitar 5.000€ de **A**

2ª Operação: creditar 5.000€ em **B**

Se ocorrer uma falha de sistema entre a primeira e a segunda operações, qual será o resultado? O montante de 5.000€, retirado de **A** nunca é creditado em **B**. A base de dados fica corrompida sem que o sistema se aperceba.

Daqui se conclui da necessidade de utilização de transações. Utilizando transações, as operações sobre as bases de dados devem ser executadas como um todo, de modo a produzirem um resultado válido. Ou seja, se falhar uma operação intermédia, as operações anteriores são anuladas e o sistema volta ao estado inicial.

Propriedades fundamentais das transações

As transações apresentam um conjunto de quatro propriedades fundamentais, conhecidas pela sigla **ACID** (*Atomicity*, *Consistency*, *Isolation*, *Durability*). Essas propriedades são:

- Atomicidade (*atomicity*): todas as operações que constituem a transação formam um grupo indivisível (atómico). Deverão todas ser concluídas com sucesso (*commit*). Caso falhe alguma das operações, serão anuladas todas as operações da transação que entretanto foram executadas (*rollback*);
- Integridade (*consistency*): a integridade, ou consistência, da base de dados deve ser sempre assegurada quando termina uma transação que envolva atualização de dados. Ou seja, os dados têm de estar válidos;
- Isolamento (*isolation*): o sistema deve garantir que cada transação (caso estejam em execução simultânea várias transações sobre os mesmos dados) não irá interferir nem ser interferida por outras transações;
- Persistência (*durability*): após uma transação terminar com sucesso (*commit*), as suas atualizações sobre a base de dados passam a ser efetivas, sobrevivendo a qualquer tipo de falhas que eventualmente possam ocorrer.

Comandos SQL

No SGBD MySQL as instruções básicas para trabalhar com transações são START TRANSACTION e COMMIT.

Por exemplo para efetuar uma transferência bancária entre duas contas:

```
-- Importante: começar por desativar o mecanismo de autocommit
SET autocommit = 0;

START TRANSACTION;
UPDATE Contas SET saldo = saldo - 5000 WHERE ID = 98;
UPDATE Contas SET saldo = saldo + 5000 WHERE ID = 7;
COMMIT;
```

O comando START TRANSACTION inicia uma transação e o comando COMMIT conclui uma transação.

10. Triggers

Triggers são procedimentos de código, na base de dados, executados de forma automática quando uma determinada condição acontece.

Apesar das implementações variarem de acordo com o SGBD utilizado, de uma forma geral, os *triggers* são causados por uma das seguintes operações:

- comandos INSERT, UPDATE ou DELETE numa tabela ou vista
- comandos CREATE, ALTER ou DROP num objeto de uma base de dados (tabelas, vistas, etc.)
- eventos da base de dados

Em que situações concretas são utilizados *triggers*?

São utilizados em várias situações, tais como:

- armazenar dados de histórico, que serão apagados/alterados sempre que existem operações introdução, edição ou eliminação
- validar dados antes de ser efetuada uma inserção ou edição de dados
- atualizar cálculos em resposta a alterações nos dados

11. Apoio à Administração de Bases de Dados

A administração de uma base de dados é feita através de, entre outros, os comandos GRANT e REVOKE:

- GRANT: atribui permissões a utilizadores
- REVOKE: retira permissões a utilizadores

As permissões que um utilizador pode ter ou não, são:

- SELECT: a possibilidade de consultar dados em tabelas
- INSERT: a possibilidade inserir novos registos em tabelas
- UPDATE: a possibilidade de modificar registos
- DELETE: a possibilidade de eliminar registos
- REFERENCES
- USAGE

GRANT

Esta instrução concede permissões a utilizadores específicos.

Sintaxe (simplificada):

```
GRANT {Lista de permissões | ALL PRIVILEGES}
ON Tabela
TO {Utilizadores | PUBLIC}
```

Componentes:

`Lista de permissões`	Uma ou mais permissões: SELECT DELETE INSERT UPDATE REFERENCES USAGE
`Tabela`	A tabela na qual se aplica(m) a(s) permissão(ões)
`Utilizadores`	Os utilizadores a quem se aplicam as permissões

Exemplo: Atribuir ao utilizador Paulo a permissão para efetuar consultas na tabela Clientes

```
GRANT SELECT
ON Clientes
TO Paulo;
```

REVOKE

Esta instrução retira permissões a utilizadores específicos.

Sintaxe (simplificada):

```
REVOKE {Lista de permissões | ALL PRIVILEGES}
ON Tabela
FROM {Utilizadores | PUBLIC}
```

Componentes:

`Lista de permissões`	Uma ou mais permissões: SELECT DELETE INSERT UPDATE REFERENCES USAGE
`Tabela`	A tabela à qual se retiram as permissões
`Utilizadores`	Os utilizadores a quem se retiram as permissões

Exemplo: Retirar ao utilizador Paulo a permissão para efetuar consultas na tabela Clientes

```
REVOKE SELECT
ON Clientes
FROM Paulo;
```