

Módulo

Bases de Dados

Curso

OutSystems


A linguagem SQL

doc v 2.5



```
1  SELECT *
2  FROM customers
3  WHERE favorite_website = 'techonthenet.com'
4  ORDER BY last_name ASC;
```

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
9000	Johnson	Derek	techonthenet.com

A linguagem SQL (*Structured Query Language*)

Módulos da linguagem:

- **DDL**, *Data **Definition** Language*

Inclui um conjunto de comandos para criação de tabelas, regras de integridade de dados, *views*, etc., assim como para alteração ou remoção.

- **DML**, *Data **Manipulation** Language*

Inclui instruções para efectuar interrogações sobre bases de dados, bem como para inserir, alterar ou remover registos. As instruções de leitura são uma implementação da Álgebra Relacional.

- **DCL**, *Data **Control** Language*

Inclui instruções para definição de utilizadores, grupos de utilizadores e permissões de acesso às tabelas.

- **TCL**, ***Transaction** Control Language*

Inclui instruções para programação de transacções (sequências de instruções indissociáveis).

- **PSM**, ***Persistent Stored Modules***

Permite programar e executar programas guardados na base de dados.
Existem dois tipos de módulos: *stored procedures* e *triggers*).

Regras sintáticas gerais da linguagem

Não há distinção entre **maiúsculas e minúsculas**

Nestes slides, as palavras-chave do SQL são colocadas em maiúsculas apenas para realçar essas palavras.

Comentários são iniciados por dois hífenes (--) e são válidos apenas até ao fim da linha. Os delimitadores “//” e “/* */”, típicos de muitas linguagens de programação, também são aceites.

Exº: SELECT * FROM Alunos -- Lista todos os alunos.
 // Também se pode comentar assim.
 /* E os comentários de várias linhas
 são delimitados assim. */

Dentro de um *script* de SQL, o **separador de instruções** é o ponto-e-vírgula.

Exº: INSERT INTO Países VALUES ('Portugal');
 INSERT INTO Países VALUES ('Espanha');

Se o nome de uma tabela ou coluna possuir **caracteres acentuados, pontuação** ou **espaços** em branco, deve ser colocado dentro de **...`** (SQL standard) ou **[...]** (no dialeto T-SQL).

Exº: SELECT * FROM `Posição em campo` (SQL standard)
 ou ... FROM [Posição em campo] (T-SQL)

Valores literais em SQL

O **texto** literal deve ser colocado entre apóstrofos:

```
SELECT * FROM Presidentes WHERE nome = 'Joe Biden'
```

Valores **numéricos** não possuem delimitadores:

```
SELECT * FROM Clientes WHERE idade < 40
```

Separador decimal é o ponto:

```
SELECT código, preço * 200.412 FROM Produtos
```

Datas e tempos são colocados entre apóstrofos:

```
Exº: SELECT * FROM Exames WHERE data > '2021/06/1'
```

```
Exº: SELECT * FROM Exames WHERE hora > '14:00'
```

```
Exº: SELECT * FROM Exames WHERE data_hora = '8/6/2021 14:00'
```

Funções de texto

Concatenação:

- Operador '||' e a função **CONCAT**
 - 'Exmo. Sr. ' || nome_cliente
 - CONCAT ('Exmo. Sr. ', nome_cliente)
 - MySQL / MariaDB
 - não suportam o operador '||'.
 - Tem que se usar a função CONCAT(...);
- Alguns sistemas usam também o operador + :
 - 'Exmo. Sr. ' + nome_cliente → MS SQL Server, MS Access

Quantidade de caracteres num valor textual: LENGTH ()

Há também funções para obter partes de uma string, tipicamente: left (...), right (...), mid (...), ...).

Os nomes e parâmetros destas funções variam com o sistema de BD, deve consultar-se o manual deste.

Mais informação: <https://mariadb.com/kb/en/string-functions/>

Funções de Datas e Tempos

Obtenção de data e hora actual, varia entre sistemas:

- **CurDate()** ou **current_date()** em MySQL / MariaDB;
- **GetDate()** em MS SQL Server
- **Current_date** ou **Sysdate** em Oracle;

Algumas destas funções, apesar da denominação apenas mencionar “date”, retornam também informação acerca da hora actual.

Obter componentes da data/hora:

- **DAY** (data_venda): o dia do mês;
- **MONTH** (data_venda): o número do mês;
- **YEAR** (data_venda): o ano
- **WEEKDAY** (data_venda): o dia da semana (1=Domingo ... 7=Sábado)
- **hour** (hora_venda): a hora
- etc.

Há grande variação nos nomes e parâmetros destas funções, de sistema para sistema. Deve consultar-se o manual do sistema que está a ser utilizado.

Mais funções: <https://mariadb.com/kb/en/date-time-functions/>

Tipos de Dados SQL

Textuais

- Char [(n)] / Varchar (n)
- Long Varchar - Não impõe limites ao número de caracteres
- Text – igual a Long Varchar mas admite NULL

Booleano

- Bit / Boolean / Bool / Tinyint(1)
 - Valores: 0 e 1.

Datas e tempos

- Date
- Time
- Datetime e Timestamp
(MS SQL Server: Datetime e Datetimeoffset)

→ <https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

→ <https://docs.microsoft.com/en-us/sql/t-sql/data-types>

→ <https://www.w3resource.com/sql/data-type.php>

Numéricos

- Integer / Int
 - 4 bytes
 - -2,147,483,647 » + 2,147,483,647
- Smallint
 - 2 bytes.
 - -32,767 » +32,767 ou 0 » 65535 (Smallint Unsigned)
- Tinyint
 - 1 byte.
 - 0 » 255.
- Decimal (M[, p])
 - Vírgula fixa:
 - M = Qtd máxima de dígitos;
 - p = Qtd de dígitos decimais
 - Alguns SGBD usam os sinónimos: Numeric, Real.
- Float / Double [(p)]
 - Vírgula flutuante: os valores são armazenados em notação científica, com mantissa e expoente. P.ex: 1200 é 1.2E3.
 - Só relevante para grandes números que possam ser arredondados a partir de p dígitos.
 - p = qtd máximo de dígitos da mantissa.
 - Ocupação de memória: Float: 4 bytes; Double: 8 bytes.

<https://mariadb.com/kb/en/data-types/>

Char vs. Varchar

Diferem no armazenamento:

CHAR (n) – São sempre armazenados n caracteres; se o valor armazenado tem menos de n caracteres, o restante é preenchido com espaços em branco, os quais são retirados sempre que o valor é lido.

VARCHAR (n) – São armazenados apenas os caracteres necessários, aos quais acrescentará 1 byte para registar o tamanho da *string*, ou 2 bytes se $n > 255$.

Value	CHAR (4)	Storage required	VARCHAR (4)	Storage required
' '	' '	4 bytes	' '	1 byte
'ab '	'ab '	4 bytes	'ab '	3 bytes
'abcd '	'abcd '	4 bytes	'abcd '	5 bytes
'abcdefgh '	'abcd '	4 bytes	'abcd '	5 bytes

Ao ser lido, é mostrado com espaços. Para retirar: usar função `Rtrim (valor)`.

Extraído de: <http://dev.mysql.com/doc/refman/8.0/en/char.html>

Text e Long Varchar

Armazenam até 2.147.483.647 bytes de caracteres

Não podem ser usados em:

- ✗ Cláusulas ORDER BY, GROUP BY e UNION;
- ✗ Na cláusula WHERE, excepto com a *keyword* LIKE (mas pouco eficiente);
- ✗ Joins e subqueries;
- ✗ Índices,

Excepto se o índice for de tipo *full-text*, sendo nesse caso indexadas as palavras individualmente, e não todo o valor *text* em questão. Este tipo de índices não é rentabilizado em pesquisas do tipo 'coluna = valor' nem 'coluna LIKE valor'.

- ✗ Em parâmetros de *stored procedures*;

Datetime vs. *Timestamp* / T-SQL: *Datetimeoffset*

Datetime inclui informação de datas e horas, com precisão até aos microssegundos.

Microssegundos são relevantes para aplicações tais como leilões online.

Timestamp* / *Datetimeoffset (SQL Server) guarda a mesma informação que *Datetime* e também o fuso horário (*time zone*).

É o tipo de dados a usar para BDs com operação transnacional.

<https://docs.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql>

→ <https://dev.mysql.com/doc/refman/8.0/en/datetime.html>

→ <https://dev.mysql.com/doc/refman/8.0/en/date-and-time-literals.html>

No Transact-SQL (T-SQL), a mais difundida extensão ao SQL, o tipo de dados com fuso horário designa-se ***Datetimeoffset***, e não *Timestamp*.

→ <https://docs.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql>

Date, Time, Datetime / Timestamp

Exemplo de aplicação:

- Atributo *Filme.Estreia* – Date

Quanto à data de estreia de um filme, apenas interessa registar a sua data. Não interessa a hora, a qual, inclusive, será diferente de cinema para cinema.

- Atributo *Sessão.Hora* – Time

Neste atributo pretende-se registar a que horas existe cada sessão, em cada sala de cinema. A dita sessão existe todos os dias; logo, as datas concretas não são relevantes.

- Atributo *Reserva.DiaHora* – Datetime ou Timestamp

Cada reserva precisa de ser rotulada não apenas com a hora nem apenas com o dia, mas sim para ambos.

Enum / Enumerado

```
Create table Produto_de_Vestuario (  
  ...  
  tamanho ENUM ('XS', 'S', 'M', 'L', 'XL', 'XXL')  
  ...)
```

Existe uma **ordenação implícita** entre os valores de um enumerado, definida pela ordem com que são especificados.

Assim, no exemplo, é possível solicitar listagens de produtos ordenados por tamanho ou elaborar consultas que para visualizar apenas produtos acima ou abaixo de um determinado tamanho, entre dois tamanhos, etc.

Se em vez de enumerado for usado um tipo textual (*char*, *varchar*) só estará disponível a ordenação alfabética.

Domínios / Data Types (criados)

Novos tipos de dados podem ser definidos

`CREATE { DOMAIN | TYPE } [AS] <nome-domínio> <datatype>;`

- `CREATE TYPE dm_Morada VARCHAR(100);`
- `CREATE TYPE dm_Dinheiro Numeric (9,2);`
- `CREATE TYPE dm_Preço Numeric (5,2);`

Indicado para quando vários atributos partilham a mesma definição de tipo de dados

- Ex^os:
 - Domínio Morada para as colunas:
 - `Cliente.morada`, `Sucursal.morada`, `Encomenda.morada_entrega`;
 - Domínio Tamanho_de_Vestuario para as colunas:
 - `Produto_vestuário.Tamanho`, `Cliente.Tamanho_vestuario`
- Objectivo: facilitar a manutenção da base de dados



Criação de Tabelas

Comando para criar uma tabela:

```
CREATE TABLE <nome-da-tabela> (
    <definição-das-colunas>,
    <restrições-de-integridade> )
```

Exemplo:

```
CREATE Datatype dm morada VARCHAR(100);
```

```
CREATE TABLE Cliente (
```

```
{ cod_cliente  INTEGER NOT NULL,
  bi           INTEGER NOT NULL,
  nome         VARCHAR(100),
  morada       dm morada,
```

```
{ CONSTRAINT prim_key PRIMARY KEY (cod_cliente),  
  CONSTRAINT cand_key UNIQUE (bi));
```

Definição das colunas

Restrições de integridade

Chave alternativa

Criação de Tabelas II

Exemplos:

```
CREATE TABLE Factura (  
  num_factura    INTEGER          NOT NULL,  
  data           DATE             NOT NULL,  
  valor          DECIMAL(10,2)    NOT NULL,  
  cod_cliente    INTEGER          NOT NULL,  
  CONSTRAINT prim_key PRIMARY KEY (num_factura),  
  CONSTRAINT for_key_cliente  
    FOREIGN KEY (cod_cliente)  
    REFERENCES Cliente (codigo)  
    ON UPDATE CASCADE  
    ON DELETE RESTRICT);
```

Chave estrangeira

```
CREATE TABLE Produto (  
  cod_produto    INTEGER          NOT NULL,  
  tipo           CHAR(2)          DEFAULT 'MP' CHECK (tipo IN ('MP','PA')),  
  Designação     VARCHAR(100),  
  CONSTRAINT prim_key PRIMARY KEY (cod_produto));
```

Valor por omissão

Restrição

Criação de Tabelas III

Exemplo:

```
CREATE TABLE Item (  
    num_fatura    INTEGER    NOT NULL,  
    num_item      INTEGER    NOT NULL,  
    quantidade    INTEGER    CHECK (quantidade > 0) NOT NULL,  
    valor         DECIMAL (8,2) NOT NULL,  
    cod_produto   INTEGER          NOT NULL,  
    CONSTRAINT    pk_item    PRIMARY KEY (num_fatura, num_item),  
    CONSTRAINT    fk_item_to_fatura  
        FOREIGN KEY (num_fatura)  
        REFERENCES Factura (num_fatura)  
        ON UPDATE CASCADE  
        ON DELETE CASCADE,  
    CONSTRAINT    fk_item_to_produto  
        FOREIGN KEY (cod_produto)  
        REFERENCES Produto (codigo)  
        ON UPDATE CASCADE  
        ON DELETE RESTRICT);
```

→ Restrição

Check constraints

As validações possíveis por via de *Check* são em geral poucas e muito simples. Na maioria dos sistemas:

- Só podem ser realizadas validações que envolvam atributos do próprio registo, valores constantes e funções de sistema determinísticas.
- Não são possíveis validações que incluam:
 - ✗ Funções de sistema não determinísticas. Por exemplo:
 - *Check data_nascimento < current_date()* não é possível.
 - ✗ *Stored functions* ou *procedures*.
- As validações que exijam este tipo de elementos têm que ser realizadas através de *triggers*.

De sistema para sistema há grande variação nas possibilidades para *Check*.

Alteração e remoção de tabelas

Comando para alterar uma tabela:

ALTER TABLE *<nome-da-tabela>*
<alterações>

Exemplo:

```
ALTER TABLE cliente  
    ADD [COLUMN] telefone VARCHAR (10),  
    DROP [COLUMN] bi,  
    CHANGE [COLUMN] Numero INT IDENTITY;
```

Comando para apagar uma tabela:

DROP TABLE *<nome-da-tabela>*

Criação e remoção de índices

Comando para criar um índice numa tabela:

```
CREATE [UNIQUE] INDEX <nome-do-índice> ON <nome-da-tabela>  
( <coluna> [ASC | DESC], ... )
```

Exemplo:

```
create unique index idx_pkey on Medicamentos_em_receita (  
    Codigo,  
    ID_Receita )
```

Comando para apagar um índice:

```
DROP INDEX <nome-do-índice>
```



Notas finais

É conveniente ter um ficheiro com a definição completa da base de dados. Esse ficheiro pode ser executado sempre que seja necessário reconstruir a base de dados.

Instrução ***Insert***

Instrução ***Update***

Instrução ***Delete***

Instrução ***Select***

- *Select* (cláusula)
- *From*
- *Joins*
- *Where*
- *Order By*
- *Group By, Having* e funções de agregação
- *Union*
- *Sub queries*

Views

Insert

Instrução para inserir **novas linhas** numa tabela.

Para inserir **uma linha**:

Sintaxe:

```
INSERT INTO tabela (colunas, a, preencher)  
  VALUES (valores, a inserir, na nova linha)
```

Exº:

```
INSERT INTO Produto (cod_produto, tipo)  
  VALUES (123, 'MP')
```

Para inserir **várias linhas**:

Sintaxe 1:

```
INSERT INTO tabela (colunas, a, preencher)  
  VALUES (valores, para, uma linha),  
          (valores, para, outra linha),  
          ...  
          (valores, para, a última linha)
```

Exº:

```
INSERT INTO Produto (cod_produto, tipo)  
  VALUES (123, 'MP'),  
          (456, 'CF'),  
          (789, 'MP')
```

Sintaxe 2:

```
INSERT INTO tabela_A (colunas, a, preencher)  
  SELECT colunas, ou, valores  
  FROM tabela_B ...
```

Exº:

```
INSERT INTO Produto (cod_produto, tipo)  
  SELECT cod_materia, 'MP'  
  FROM Materias_Primas
```

Instrução para **alterar dados** já existentes.

Sintaxe:

UPDATE *tabela_a_alterar*

SET

coluna₁ = novo valor₁,

coluna₂ = novo valor₂,

...

coluna_N = novo valor_N

WHERE

expressão lógica que indica as linhas a alterar

Exº:

Mudar para o 3º ano os alunos do 2º ano que têm nota positiva:

```
UPDATE alunos
```

```
SET ano = 3
```

```
WHERE ano = 2 AND nota >= 10
```

Delete

Instrução para **remover linhas**.

Sintaxe:

DELETE FROM *tabela*

WHERE *expressão lógica que indica as linhas que queremos apagar*

Apagar os clientes residentes no código postal 1200:

```
DELETE FROM Cliente  
WHERE CodPostal = 1200
```

Apaga todos os registos na tabela Cliente:

```
DELETE FROM Cliente
```


Select

Instrução para selecção de linhas de uma ou mais tabelas;

Sintaxe:

SELECT *coluna(s) a visualizar*

FROM *tabela(s) onde constam os dados envolvidos na consulta*

WHERE *expressão lógica para filtragem das linhas*

GROUP BY *critérios para produção de valores agregados*

HAVING *expressão lógica para filtragem dos valores agregados*

ORDER BY *campo(s) pelo(s) qual(is) a listagem virá ordenada*

Exemplo:

```
SELECT Nome, Morada
FROM Clientes
WHERE Cod_Postal = 1300
ORDER BY Nome
```

-- > Mostra as colunas *nome* e *morada*

-- > da tabela *Clientes*,

-- > listando apenas as linhas onde *Cod_Postal* é 1300

-- > e ordenando-as por *nome*.

Qualquer expressão sintacticamente válida pode ser argumento da cláusula SELECT.

P. ex., os seguintes comandos são válidos:

➔ `SELECT Produto, Quantidade * Preço FROM Item;`

↳ Dá duas colunas em que a segunda é o produto das colunas *quantidade* e *preço*;

➔ `SELECT 'teste' FROM Item`

↳ Se a tabela Item tiver 20 linhas, o comando lista 20 vezes a palavra “teste”.

➔ `SELECT 'Exmo(a) . Sr(a) . ' || Nome FROM Cliente`

↳ Prefixa os nomes dos clientes com o texto “Exmo(a)...”.



Podem ser atribuídos *aliases* (sinónimos) às colunas seleccionadas.

O seguinte comando permite dar um nome – *Valor* – à segunda coluna fornecida

```
SELECT Produto, Quantidade * Preço AS Valor FROM Item
```

↳ Na listagem/tabela resultante, a coluna aparece com o título “Valor”;

Se aos resultados desta instrução aplicarmos novo SELECT, este poderá referir-se aos valores *Quantidade*Preço* como *Valor*:

```
SELECT Produto, Valor * 0.05 AS Desconto, Valor - Desconto AS  
Custo  
FROM (SELECT Produto, Quantidade * Preço AS Valor FROM Item) AS t
```

↳ Irá retornar uma tabela com 3 colunas: Produto, Desconto, Custo;

A palavra “AS” é opcional:

```
SELECT nAluno Número, Nome FROM Alunos
```





Caso pretendamos visualizar todos os campos de uma tabela, como alternativa a enumerá-los todos, pode-se usar a constante *:

```
SELECT * FROM Item
```

Em consultas a várias tabelas, o '*' pode ser qualificado (prefixado) com o nome de uma das tabelas:

```
SELECT Cliente.*, localidade FROM Cliente, CodPostal ...
```

↳ Lista todas as colunas da tabela *Cliente* e a coluna localidade da tabela *CodPostal*;

Caso pretendamos eliminar duplicados na listagem obtida, usamos a cláusula **DISTINCT**.

```
SELECT DISTINCT CodPostal FROM Cliente
```

...fornece os códigos postais onde temos clientes

Caso pretendamos apenas visualizar algumas linhas de uma tabela:

```
SELECT TOP 1 * FROM Itens
```

- fornece a primeira linha da tabela Item;
- não sendo indicada qualquer critério de ordenação, será fornecida uma linha qualquer daquela tabela.

```
SELECT TOP 3 * FROM Itens
```

Conjuntamente com a cláusula *Order By*, permitem obter a(s) primeira(s) linha(s) de acordo com determinado critério.

```
SELECT TOP 5 *  
FROM Alunos  
ORDER BY Idade
```

- fornece os 5 alunos mais novos

Em **MySQL** e **MariaDB**, *Top* é substituído por **Limit** e este é indicado no fim da instrução.

Ex:

```
SELECT *  
FROM Itens LIMIT 3
```

```
SELECT *  
FROM Alunos  
ORDER BY Idade LIMIT 5
```



Caso se pretenda trabalhar com duas colunas com o mesmo nome (necessariamente pertencentes a tabelas distintas) é necessário preceder a coluna com tabela a que pertence

```
SELECT Factura.Nr, Linha.Nr FROM Factura, Linha ...
```

...Ambas as tabelas consultadas (*Factura* e *Linha*) possuem um atributo *Nr*.

Caso contrário, o interpretador de SQL considera que há ambiguidade e dá erro.

Na cláusula *From* indicam-se os nomes das tabelas envolvidas na consulta, separadas por vírgulas.

Quando indicadas duas tabelas, o SQL combina todas as linhas de uma tabela com todas as da outra. Se indicadas três tabelas, combina a tabela 1 com a 2 e o resultado com a 3. Esta operação designa-se **produto cartesiano** ou **junção cruzada** / **cross join** de tabelas.

Exemplos:

SELECT * FROM Produto, Região

Cruza todos os produtos com todas as regiões. Útil para realizar um relatório.

SELECT * FROM Equipa, Equipa

Cruza todas as equipas com todas as equipas, o que, numa base de dados futebolística, corresponde aos jogos de um campeonato.

Seria, no entanto, necessário excluir os jogos de cada equipa com ela própria (ver: cláusula *Where*)

O * (asterisco) fará aparecer todas as colunas de ambas as tabelas.

Normalmente, ao pretendermos cruzar dados de duas tabelas, não será um produto cartesiano o que pretendemos realizar, mas sim uma **junção / *join*** (termo mais comum, mesmo em Português).

Um join é a colagem lado a lado das linhas de uma tabela com as de outra, em que a selecção de quais linhas juntam com quais obedece a um critério indicado. O critério mais frequente é a igualdade entre os valores de uma chave estrangeira e os da respectiva chave primária.

```
SELECT * FROM Cliente JOIN Localidade
```

```
SELECT * FROM Cliente LEFT OUTER JOIN Localidade
```

Existem vários critérios de *join*:

- **Key Join:** SELECT * FROM Cliente **JOIN** Localidade
 - O critério é a igualdade dos valores nas colunas ligadas por uma chave estrangeira.
 - É necessário que exista uma – e apenas uma – chave estrangeira a ligar as duas tabelas.
- **Natural Join:** SELECT * FROM Cliente **NATURAL JOIN** Localidade
 - Critério: igualdade de valores nas colunas que possuem o mesmo nome e com tipos de dados compatíveis em ambas as tabelas.
- **Join on expression:** SELECT * FROM Cliente **JOIN** CPostal
 ON Cliente.CodPostal = CPostal.Cod4 || '-' || CPostal.Cod3
 - Critério: fornecido no comando através de uma expressão lógica. É o mais flexível.
 - Alguns sistemas só trabalham com esta forma de *join*.

Usando apenas *Join*, sem indicar o tipo de critério, é realizado um *key join*:

Join = Key Join

As diferentes possibilidades num *join* quanto ao tratamento das linhas de uma das tabelas que não ligam a qualquer linha da outra podem ser aplicadas com:

INNER

LEFT [OUTER]

RIGHT [OUTER]

FULL [OUTER]

Os quais podem ser conjugados com *Key Join*, *Natural Join* ou *Join-On*. Ex^ºs:

```
SELECT * FROM Cliente NATURAL INNER JOIN Localidade;
```

```
SELECT * FROM Cliente NATURAL LEFT OUTER JOIN Localidade;
```

```
SELECT * FROM Cliente KEY RIGHT JOIN Localidade;
```

```
SELECT * FROM Cliente LEFT OUTER JOIN Localidade  
ON Cliente.Cod_Postal = Localidade.Cod_Postal;
```

Quando nada é indicado, é realizado um *inner join*.



À semelhança dos sinónimos dos atributos, é possível atribuir *aliases* (sinónimos) às tabelas mencionadas na cláusula FROM:

Sintaxe: SELECT ... FROM <Tabela> [AS] <Sinónimo> ...

Exemplo: SELECT * FROM Docente **AS** prof

A palavra “AS” é opcional: SELECT * FROM Docente **prof**

Os sinónimos no *From* são habitualmente usados com abreviaturas para simplificar instruções:

```
SELECT p.Codigo, Nome
      FROM Produto p, Encomenda e
      WHERE p.codigo = e.codigo ...
```

São essenciais em alguns *sub-queries* (a ver mais adiante).

Na cláusula *WHERE* pode constar qualquer expressão lógica. A expressão é avaliada linha a linha, isto é, para cada linha o SQL avalia o valor da expressão e, caso seja verdadeira, *devolve* a linha.

- `SELECT * FROM Cliente WHERE profissao = 'Médico'`
 - Os apóstrofos são importantes; não sendo indicados, *Médico* será incorrectamente interpretado como o nome de um atributo da tabela consultada.
- `SELECT * FROM Cliente WHERE CodPostal >= 1000 AND CodPostal < 2000`
- `SELECT * FROM Cliente WHERE (profissao = 'Médico' OR profissao = 'Engenheiro') AND CodPostal = 3000`
- `SELECT * FROM CodPostal WHERE 1 = 1`
o mesmo que
`SELECT * FROM CodPostal`
→ Devolve todos os registos

Principais operadores utilizados na cláusula *WHERE*:

=, <, >, >=, <=, <>, AND, OR, NOT, IN, LIKE, BETWEEN e IS NULL

O operador **IN** é verdadeiro quando um elemento faz parte de um conjunto.

- `SELECT * FROM Cliente WHERE Nacionalidade IN ('Portugal', 'Brasil', 'Angola')`
 → Todos os clientes portugueses, brasileiros ou angolanos.
- Simplificação de:
 campo = 'valor1' OR campo = 'valor2' OR ...

O operador **LIKE** permite a utilização de *wildcards*.

- `SELECT * FROM Cliente WHERE Nome LIKE 'João%'`
 → Todos os clientes começados por “João”, incluindo os clientes unicamente chamados “João”.
- `SELECT * FROM Carro WHERE modelo LIKE 'Audi A_'`
 → Todos os carros cujo modelo começa com ‘Audi A’ seguido de um carácter (qualquer).

O operador **NOT** pode anteceder uma condição ou um operador:

```
SELECT * FROM Cliente  
WHERE Nacionalidade NOT IN ('Portugal', 'Brasil')
```

→ Todos os clientes excepto portugueses e brasileiros.

O operador **IS NULL** permite lidar com campos não preenchidos.

```
SELECT * FROM Cliente WHERE profissão IS NULL
```

→ Todos os clientes com profissão desconhecida

```
SELECT * FROM Cliente WHERE Nacionalidade IS NOT NULL
```

→ Todos os clientes com nacionalidade conhecida

Simplificação de “campo $\geq X$ AND campo $\leq Y$ ”:

```
SELECT * FROM CLIENTE WHERE CodPostal BETWEEN 1000 AND 2000
```

A cláusula *WHERE* pode ser utilizada para produzir *joins*. Os dois comandos seguintes produzem o mesmo resultado.

```
SELECT * FROM Cliente INNER JOIN Localidade  
    ON Cliente.Cod_Postal = Localidade.Cod_Postal;
```

```
SELECT * FROM Cliente, Localidade  
    WHERE Cliente.Cod_Postal = Localidade.Cod_Postal;
```