Módulo

Bases de Dados

Curso

OutSystems

A linguagem SQL

doc v 2.5

José Farinha
2022

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 sintácticas 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 Paises VALUES ('Portugal');
INSERT INTO Paises 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:

```
Ex°: SELECT * FROM Exames WHERE data > '2021/06/1'
Ex°: SELECT * FROM Exames WHERE hora > '14:00'
Ex°: 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
  - Valores: 0 e 1.
  - Em alguns SGBD: Boolean.

Datas e tempos

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

➔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 (Unsigned Smallint)
- 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.
  - *p* = qtd máximo de dígitos da mantissa.
  - Requisitos de armazenamento:
    - Float: 4 bytes;
      ou 8 bytes, se *p* > 24.
    - Double: 8 bytes.

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 acrescerá 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.

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

- Exºs:
  - 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_factura    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_factura, num_item),
     CONSTRAINT     fk_item_to_factura
          FOREIGN KEY (num_factura)
          REFERENCES Factura (num_factura)
          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.

Índice

Instrução ***Select***

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

Instrução ***Insert***

Instrução ***Update***

Instrução ***Delete***

Views

Comando SELECT

Tipo de instrução para selecção de linhas de uma ou mais tabelas;

O resultado de um comando *Select* é também uma tabela;

Outras designações para um comando *Select*:

Query

Pesquisa / Procura

Consulta

Comando SELECT

Um comando SQL para selecção de linhas obedece à estrutura:

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 Cliente
WHERE Cod_Postal = 1300
ORDER BY Nome;
```

☞ Lista o nome e morada de uma tabela de clientes, mas apenas os clientes cujo código postal seja 1300, ordenando por nome.

Nota

É importante notar que qualquer comando SELECT *fornece* uma tabela (um conjunto de colunas e linhas). Mesmo sendo temporária, esta tabela está dotada das mesmas possibilidades para consulta que as tabelas originais da base de dados;

Sempre que, no contexto da sintaxe da instrução *Select* for referida uma tabela, esta referência deve ser interpretada num sentido lato: uma tabela *original* (definida no esquema relacional) ou o resultado de um comando *select*.

Exº:

```
SELECT Nome, Morada
FROM ( SELECT Nome, Morada, Facturacao
       FROM Cliente
       WHERE Cod_Postal = 1300 ) as t
WHERE Facturacao > 100000
ORDER BY Nome;
```

Tabela

Insert

Através do comando INSERT podem-se inserir uma ou várias linhas em simultâneo.

Para inserir uma linha:

INSERT INTO <*tabela_a_inserir*> (<*colunas_onde_devem_ser_inseridos_os_valores*>)
VALUES (<*valores_a_inserir*>)

Exº:

INSERT INTO Produto (cod_produto, tipo) VALUES (123456, 'MP');

Para inserir um conjunto de linhas:

INSERT INTO <*tabela_a_inserir*> (<*colunas_onde_devem_ser_inseridos_os_valores*>)
SELECT <*valores_a_inserir*> **FROM ...**

Exº:

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

Update

O comando UPDATE altera dados já existentes.

Sintaxe:

UPDATE <*tabela_a_alterar*>

SET <*coluna_a_alterar*> = <*expressão*> [, <*coluna2*> = <*expressão2*> …]

WHERE <*expressão_lógica_que_indica_as_linhas_a_alterar*>

Mudar os alunos do 2º ano para o 3º ano:

```
UPDATE Aluno
  SET ano_curricular = 3
  WHERE ano_curricular = 2
```

Delete

O comando DELETE remove 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;
```

Cláusula *Select*

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)…”.

Cláusula *Select* – *Utilização de Sinónimos*

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
```

Cláusula *Select* – * (asterisco)

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*;

Cláusula *Select – Distinct*

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

```
SELECT DISTINCT CodPostal FROM Cliente
```

…fornece os códigos postais onde temos clientes

Cláusula *Select* – *First* e *Top*

Caso pretendamos apenas visualizar algumas linhas de uma tabela:

```
SELECT TOP 1 * FROM Item
```

- 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 Item
```

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

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

- fornece os 5 alunos mais novos

Instrução Select

Cláusula *Select* – Prefixos

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.

Instrução Select

Cláusula *From* – Produto cartesiano

Na cláusula FROM indicam-se os nomes das tabelas envolvidas na interrogação, separadas por vírgulas.

Quando existe mais que uma tabela, o SQL executa um produto cartesiano entre as 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 uma competição. Nota: seria, no entanto, necessário excluir o jogo de cada equipa com ela própria (ver: cláusula *Where*)

O * (asterisco) fará aparecer todos os atributos de ambas as tabelas.

Instrução *Select*

Cláusula *From – Join*

Normalmente, ao pretendermos cruzar dados de duas tabelas, não será um produto cartesiano que pretendemos realizar, mas sim um *Join*.

SELECT * FROM Cliente **JOIN** Localidade

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

Instrução Select

Cláusula *From* – Critérios de *Join*

Existem vários tipos de *join*, variando no critério que estabelece a correspondência entre as linhas das tabelas relacionadas:

- ***Key Join***: SELECT * FROM Cliente **KEY JOIN** Localidade
  - O critério é a igualdade dos valores nas colunas das duas tabelas ligadas por 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 nos atributos 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 do 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

Instrução *Select*

Cláusula *From – Inner* e *Outer Joins*

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*. Exºs:

```
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*.

Instrução Select

Cláusula *From* – *Aliases* (sinónimos)

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).

Instrução *Select*

Cláusula *From* – Tabela *Dummy*

Existe uma tabela de sistema denominada *DUMMY*, que apenas contém uma linha. Pode ser utilizada quando pretendemos listar uma expressão que não é obtida a partir de nenhuma tabela.

Exº: Se necessitarmos de obter o valor de π para usar em SQL, podemos escrever:

```
SELECT ACOS(1) FROM DUMMY;
```

Devolve o arco cujo coseno é 1.

Instrução *Select*

Cláusula *Where*

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

Instrução *Select*

Cláusula *Where* – Operadores lógicos

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 caracter (qualquer).

Instrução *Select*

Cláusula *Where* – Operadores lógicos

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 >= X AND campo <= Y”:

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

Instrução *Select*

Cláusula *Where* – Realização de *Joins*

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;
```

Instrução *Select*
GROUP BY, HAVING e funções de agregação

Recorrendo apenas às cláusulas anteriores não é possível obter tabelas com valores estatísticos (somatórios, médias, etc.). Tal acontece porque as operações de agregação (que envolvem vários registos) não poderem ser calculadas linha a linha.

Por exemplo, o comando para listar os códigos postais onde há mais que dois clientes **não pode** ser efectuado da seguinte forma:

```
SELECT CodPostal FROM Cliente WHERE COUNT(CodPostal) > 2
```

O comando é incorrecto porque a cláusula WHERE é testada linha a linha e numa linha não é possível obter o total de códigos postais.

Instrução *Select*
GROUP BY, HAVING e funções de agregação

As cláusulas GROUP BY e HAVING permitem manipular valores agregados.

- A cláusula GROUP BY permite a definição de grupos.
- A cláusula HAVING é equivalente à cláusula WHERE, mas inclui expressões lógicas relativas aos agrupamentos criados pela cláusula GROUP BY.

O exemplo anterior encontra-se bem formulado com o seguinte comando:

```
SELECT CodPostal FROM Cliente
    GROUP BY CodPostal
    HAVING COUNT(CodPostal) > 2
```

- A cláusula GROUP BY agrupa os clientes por código postal
- A cláusula HAVING selecciona os grupos cujo número de elementos é superior a 2.

Instrução *Select*

GROUP BY, HAVING e funções de agregação

O seguinte comando lista, para cada código postal, o número de clientes que a ele estão associados, desde que exista mais do que um cliente:

```
SELECT CodPostal, COUNT(*)
    FROM Cliente
    GROUP BY CodPostal
```

Cliente

Número	*Nome*	*CodPostal*
001	João	1500
013	Ana	2100
056	Luis	1500
001	Paula	2100
011	Nuno	1300

CodPostal = 1500 (Total:2)

Número	*Nome*	*CodPostal*
001	João	1500
056	Luis	1500

CodPostal = 2100 (Total:2)

Número	*Nome*	*CodPostal*
013	Ana	2100
001	Paula	2100

CodPostal = 1300 (Total:1)

Número	*Nome*	*CodPostal*
011	Nuno	1300

Nova tabela temporária

CodPostal	*COUNT (*)*
1500	2
2100	2
1300	1

Instrução *Select*

GROUP BY, HAVING e funções de agregação

O seguinte comando lista, para cada código postal, o número de clientes que a ele estão associados, desde que exista mais do que um cliente:

```
SELECT CodPostal, COUNT(*)
    FROM Cliente
    GROUP BY CodPostal HAVING COUNT(*) > 1
```

Cliente

Número	*Nome*	*CodPostal*
001	João	1500
013	Ana	2100
056	Luis	1500
001	Paula	2100
011	Nuno	1300

CodPostal = 1500 (Total:2)

Número	*Nome*	*CodPostal*
001	João	1500
056	Luis	1500

CodPostal = 2100 (Total:2)

Número	*Nome*	*CodPostal*
013	Ana	2100
001	Paula	2100

CodPostal = 1300 (Total:1)

Número	*Nome*	*CodPostal*
011	Nuno	1300

Nova tabela temporária

CodPostal	*COUNT (*)*
1500	2
2100	2
1300	1

Having
Count (*) > 1

Resultado Final:

CodPostal	*COUNT(*)*
1500	2
2100	2

Sempre que existe um GROUP BY na instrução, todos elementos da cláusula *Select* têm que ser funções de agregação ou atributos incluídos na cláusula GROUP BY.

Instrução *Select*
Funções de agregação

Para além da função **COUNT** e, existem outras, tais como **SUM** (somatório), **AVG** (média), **MAX** e **MIN** (valor mais alto e mais baixo).

Apenas a função COUNT não necessita de argumento:

O primeiro comando retorna o total de registos de clientes enquanto o segundo devolve o total de clientes com o código postal preenchido:

```
SELECT COUNT(*) FROM Cliente;
SELECT COUNT(CodPostal) FROM Cliente;
```

Note-se que o segundo comando é equivalente ao seguinte:

```
SELECT COUNT(*) FROM Cliente WHERE CodPostal IS NOT NULL
```

SELECT – *Subqueries*

Uma *subquery* é um comando SELECT dentro de um comando SELECT. Muitas interrogações apenas podem ser resolvidas utilizando *subqueries*.

Um comando SELECT normalmente "liga-se" a outro através da cláusula WHERE.

Os operadores usados para conjugar uma *query* com uma *subquery* na cláusula WHERE daquela:

- **IN**
- **EXISTS**
- **ALL**
- **ANY**

Subqueries – O operador *IN* (I)

Sintaxe:

SELECT ... FROM tabela1 WHERE atributo1 IN (SELECT atributo2 FROM ...)

Devolve *True* para os registos da tabela1 que possuem em atributo1 um valor pertencente ao conjunto de valores seleccionados pela *subquery*;

Exº: Jogadores que já marcaram golos:

```
SELECT Nome FROM Jogador
  WHERE nrJogador IN (SELECT marcador FROM Golo)
```

Nota: É equivalente a:

```
SELECT DISTINCT (Nome) FROM Jogador, Golo
 WHERE Jogador.nrJogador = Golo.marcador;
```

isto é:

```
SELECT DISTINCT (Nome) FROM Jogador JOIN Golo
```

O operador *IN* apenas pode ser usado com *subqueries* com um só atributo na cláusula *SELECT*;

Subqueries – O operador *IN* (II)

Como qualquer operador lógico, o *IN* pode ser conjugado com o *NOT*.

Ex.º: Jogadores que ainda não marcaram golos:

```
SELECT Nome FROM Jogador
  WHERE nrJogador NOT IN (SELECT marcador FROM Golo)
```

Nota: É equivalente a:

```
SELECT DISTINCT (Nome)
  FROM Jogador LEFT OUTER JOIN Golo
  WHERE Golo.marcador IS NULL
```

Subqueries – O operador *IN* (III)

Um *subquery* pode referir a tabela consultada no *query* exterior:

Exº: Jogadores que já marcaram golos a jogar fora:

```
SELECT Nome FROM Jogador
   WHERE nrJogador IN
      (SELECT marcador FROM Golo
        WHERE Jogador.clube = Golo.clubeVisitante
       );
```

Query exterior

Sub Query

Conceptualmente, uma *query* é um processo que percorre todas as linhas da tabela indicada em FROM e que avalia a condição WHERE para cada uma dessas linhas.

Consequentemente, se existir uma *subquery* na cláusula WHERE, essa *subquery* é executada uma vez para cada linha da tabela exterior (no exemplo: Jogador).

A cada execução da *subquery*, a referência à tabela exterior é substituída pelo registo dessa tabela exterior que está a ser processado naquele momento. No exemplo:

- Na 1ª execução da *subquery*: Jogador ≡ 1º jogador da tabela
- Na 2ª execução da *subquery*: Jogador ≡ 2º jogador da tabela
- ...

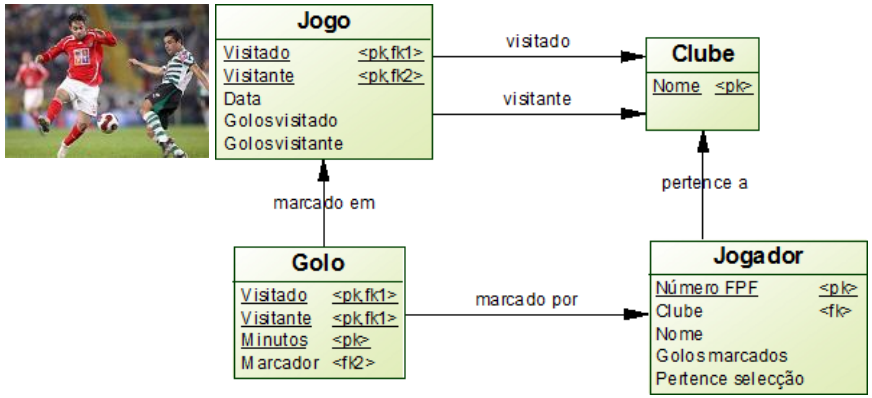
Subqueries – O operador *EXISTS* (I)

Sintaxe:

SELECT ... FROM tabela1 WHERE EXISTS (SELECT * FROM ...)

Dá *True* se a *subquery* retorna pelo menos uma linha;

Exº:



Jogadores que já marcaram golos:

```
SELECT Nome FROM Jogador
 WHERE EXISTS (SELECT * FROM Golo
                        WHERE marcador = Jogador.nrJogador)
```

referência para o registo a ser analisado exteriormente

Subqueries – O operador *EXISTS* (II)

Quase sempre, a *subquery* dentro do *EXISTS* faz referência à tabela processada na *query de fora*.

É pouco natural que não o faça.

Sintaxe:

SELECT ... FROM *tabela1* WHERE **EXISTS (**

SELECT * FROM ... WHERE ... *tabela1*...

)

Para simplificação, basta que a cláusula *SELECT* da *subquery* dentro do *EXISTS* tenha apenas o símbolo *.

Há quem prefira escrever "SELECT 1 …".

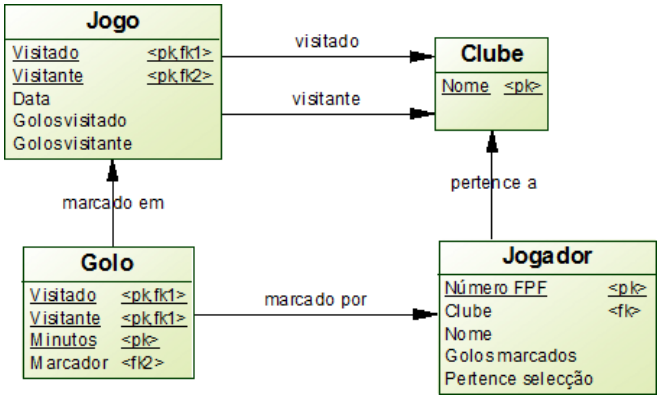
A execução do EXISTS (SELECT …) é indiferente ao que surge depois de "SELECT".

Subqueries – O operador *EXISTS* (III)

Como qualquer operador lógico, o EXISTS pode ser conjugado com o NOT:

Ex.º: Jogos em que não houve golos:

```
SELECT * FROM Jogo as J
   WHERE NOT EXISTS
            (SELECT * FROM Golo as G
               WHERE G.visitado  = J.visitado AND
                     G.visitante = J.visitante)
```



Subqueries – reflexivos

Quando:

- o *subquery* é sobre a mesma tabela que o *query* exterior
- e se pretende relacionar cada registo processado pelo SELECT exterior (*query*) com os registos processados pelo SELECT interior (*subquery*):

➔ torna-se necessário usar **sinónimos distintos** para cada uma das duas referências ao nome da tabela

Ex.º: Para listar os jogadores com mais golos de cada equipa:

```
SELECT Nome FROM Jogador J1   (ou: Jogador AS J1)
WHERE NOT EXISTS
      (SELECT * FROM Jogador J2   (ou: Jogador AS J2)
       WHERE J2.Golos_marcados > J1. Golos_marcados
AND
                J2.equipa = J1.equipa)
```

Subqueries – Os operadores *ALL* e *ANY* *(I)*

Os operadores ALL e ANY permitem comparar um valor *x* com um conjunto de valores *Cx*:

ALL: Fornece TRUE se a comparação for verdadeira entre o valor *x* e ***todos*** os elementos do conjunto *Cx*.

ANY: Fornece TRUE se a comparação for verdadeira entre o valor *x* e ***pelo menos um*** dos elementos do conjunto *Cx*.

A comparação pode basear-se em qualquer um dos operadores lógicos: =, <>, >, <, >=, <=

x **<= ALL** (*Cx*)

x **> ANY** (*Cx*)

Usam-se muito com *subqueries*.

Subqueries – Os operadores *ALL* e *ANY* *(II)*

Exemplos:

– Comando para retornar o melhor marcador:

```
SELECT Nome FROM Jogador
 WHERE
      Golos_marcados >= ALL ( SELECT Golos_marcados
                                FROM Jogador )
```

– Jogadores que não pertencem à Selecção mas que marcaram mais golos do que alguns da selecção:

```
SELECT Nome FROM Jogador
 WHERE
    pertence_selecção = FALSE AND
    Golos_marcados > ANY ( SELECT Golos_marcados
                             FROM Jogador
                             WHERE
                             pertence_selecção = TRUE )
```

Subqueries – Nas cláusulas *SELECT* e *FROM*

As *subqueries* também podem ser colocadas nas cláusula *SELECT* e *FROM*.

O seguinte exemplo devolve, para cada cliente, o total de facturas associadas:

```
SELECT Nome, (SELECT COUNT(*)
              FROM Factura
              WHERE Factura.nrCliente = Cliente.nrCliente)
                                                FROM Cliente
```

(Também poderia ser realizado com *join* e *group by*)

Comando UNION

A união de comandos SELECT é efectuada através do operador UNION.

O seguinte comando devolve os nomes dos clientes e fornecedores:

```
SELECT Nome FROM Cliente
UNION
SELECT Nome FROM Fornecedor;
```

Caso não pretendamos eliminar nomes duplicados o comando será:

```
SELECT Nome FROM Cliente
UNION ALL
SELECT Nome FROM Fornecedor;
```

Views

As *views* não são mais do que comandos SELECT armazenados.

Note-se que o resultado de uma execução de uma view (os registos que ela *devolve)* depende dos registos armazenados no momento nas tabelas de suporte à view. As views podem ser utilizadas dentro de comandos SELECT.

```
CREATE VIEW Clientes_Lisboa
AS Select * FROM Cliente
  Where Localidade = 'Lisboa'
WITH CHECK OPTION;

Select Nome From Clientes_Lisboa where Idade > 30;
```

As views não podem conter a cláusula ORDER BY e apenas permitem a inserção, remoção e alteração de registos caso não contenham as cláusulas GROUP BY e UNION. A cláusula CHECK OPTION rejeita alterações e inserções na view que não obedeçam ao critério da cláusula SELECT que a define.