

---

# ***Linguagem SQL***

---

Sistemas de Informação I

---

## ***SQL (Structured Query Language) - Características***

---

- É uma linguagem standard, utilizada por um largo conjunto de SGBD relacionais para:
  - Definição de dados
  - Manipulação de dados
  - Interrogações
  - Controlo transaccional
  - Gestão de privilégios
- É um standard da ISO e da ANSI
- Existem duas formas da linguagem ser usada
  - Interactivamente
  - Embutida noutras linguagens de programação (C, C++, etc)

---

## ***SQL (Structured Query Language) - Características (cont.)***

---

- Na sua essência, é uma Linguagem não-procedimental \*
  - Especifica-se O QUÊ e não COMO
- Existe uma clara abstracção perante a estrutura física dos dados
  - não é necessário especificar caminhos de acesso, nem elaborar algoritmos de pesquisa física
- As operações efectuam-se sobre:
  - conjuntos de Dados (Tabelas)
  - não é necessário manipular os dados Linha-a-Linha.
- Não é *CASE-SENSITIVE*
- Os SGBDs podem não implementar todas as características da linguagem

\* Com as extensões passou a ser multiparadigma, possibilitando uma abordagem procedimental

---

## ***SQL – História***

---

- 1970: Codd define o Modelo Relacional
- 1974: A IBM desenvolve o SYSTEM/R com a linguagem SEQUEL
  - Mais tarde denominado SQL
- Lançamento de SGBDs comerciais
  - 1979: Primeiro SGDB comercial (Relational software Inc. hoje ORACLE Corp.)
  - 1981: SGBD INGRES
  - 1983: IBM anuncia o DB2
  - 1985: a IBM patenteou o SQL.

---

## *Normas SQL*

---

- **1986, 1987:** Norma SQL-86 (ANSI X3.135-1986 e ISO 9075:1987)
- **1989:** Norma SQL-89 (ANSI X3.135.1-1989)
- **1992: Norma SQL2** (ISO/IEC 9075:1992)
  - Revisão importante do standard
- **1999:** Norma SQL3 (ISO/IEC 9075:1999)
- **2003:** ISO/IEC 9075:2003
  - Suporte a XML
- **2006:** ISO/IEC 9075:2006
- **2008:** ISO/IEC 9075:2008

---

## ***SQL – As duas Componentes da linguagem***

---

- LDD (Linguagem de Definição de Dados)
  - Permite definir os Esquemas de Relação
  - Permite definir os atributos dos Esquemas de Relação
  - Permite definir restrições (chaves primárias, estrangeiras, etc.)
- LMD (Linguagem de Manipulação de dados)
  - Permite aceder à informação armazenada na base de dados
  - Permite inserir, eliminar e alterar a informação presente na base de dados

---

## *SQL - comandos principais*

---

- Interrogação

- SELECT

- Manipulação de Dados

- INSERT – Inserir novos registos
- UPDATE – Alterar registos existentes
- DELETE – Apagar registos

**LMD**

- Definição de Dados

- CREATE – Criar estruturas de dados (tabelas, vistas, índices)
- ALTER – Alterar estruturas de dados
- DROP – Remover estruturas de dados

**LDD**

---

## *SQL - comandos principais (cont.)*

---

- Controlo de Transacções
  - COMMIT
  - SAVEPOINT
  - ROLLBACK
- Segurança
  - GRANT – Usado para atribuir direitos de acesso
  - REVOKE – Usado para retirar direitos de acesso



---

## *Sintaxe do comando **SELECT***

---

- A sintaxe geral de uma interrogação SQL é a seguinte:  
**SELECT [DISTINCT] <colunas> | \***  
**FROM <tabelas>**  
**[WHERE <condição>]**
- Onde:
  - **<colunas>** especifica a lista de atributos cujos valores interessa conhecer
  - **<tabelas>** especifica quais as tabelas envolvidas no processamento da interrogação
  - **<condição>** é uma expressão lógica que define a condição a verificar
  - **DISTINCT** indica que se quer remover os duplicados no resultado final
  - O símbolo **\*** é utilizado quando se pretendem seleccionar todos as colunas das tabelas especificadas na cláusula **FROM**

---

## ***Operações Algébricas***

---

- Será com o comando SELECT que as operações algébricas serão implementadas
- Recordando quais são essas operações:
  - Operadores Unários
    - Restrição
    - Projecção
  - Operadores Binários
    - União
    - Intersecção
    - Diferença
    - Produto Cartesiano
    - Junção
    - Divisão

---

## Exemplo

---

- Considere-se os seguintes Esquemas de Relação e respectivas Relações
  - DEPARTAMENTO( codDept, nomeDept, localizacao )
  - CATEGORIA( codCat, designacao, salarioBase )
  - EMPREGADO( codEmp, nomeEmp, dataAdmissao, codCat, codDept, codEmpChefe )

Departamento

codDepart	nomeDepart	localizacao
1	Contabilidade	Lisboa
2	Vendas	Porto
3	Investigação	Coimbra

Categoria

codCat	designacao	salarioBase
1	CategoriaA	1.500,00 €
2	CategoriaB	1.100,00 €
3	CategoriaC	750,00 €

Empregado

codEmp	nomeEmp	dataAdmissão	codCat	codDept	codEmpChefe
1	António	20-Mar-01	1	1	1
2	João	20-Mar-01	1	2	1
3	Nuno	20-Mar-01	3	3	1
4	Carlos	6-Abr-98	3	2	2

## Operações Algébricas - Selecção ou Restrição

- **Questão:** Quais as Categorias onde o salário base é inferior a 1200€?

- Em álgebra relacional:

- $\sigma_{\text{salarioBase} < 1200} (\text{CATEGORIA})$

codCat	designacao	salarioBase
2	CategoriaB	1.100,00 €
3	CategoriaC	750,00 €

- Em SQL:

- `SELECT * FROM CATEGORIA WHERE salarioBase<1200`

- **Questão:** Quais as Categorias onde o salário base é inferior a 1000€ ou que têm a descrição 'Categoria A'?

- Em álgebra relacional:

- $\sigma_{\text{salarioBase} < 1000 \vee \text{designacao} = \text{'categoriaA'}} (\text{CATEGORIA})$

- Em SQL:

- `SELECT * FROM CATEGORIA  
WHERE salarioBase<1000 OR designacao='CategoriaA'`

codCat	designacao	salarioBase
1	CategoriaA	1.500,00 €
3	CategoriaC	750,00 €

## Operações Algébricas - Projecção

- **Questão:** Qual o nome e data de admissão de cada empregado?

- Em álgebra relacional:

–  $\pi_{\text{nomeEmp, dataAdmissão}} (\text{EMPREGADO})$

nomeEmp	dataAdmissão
António	20-Mar-01
João	20-Mar-01
Nuno	20-Mar-01
Carlos	6-Abr-98


- Em SQL:

```
SELECT nomeEmp, dataAdmissão FROM EMPREGADO
```

- **Questão:** Quais os códigos das categorias de cada um dos empregados?

- Em álgebra relacional:

–  $\pi_{\text{codCat}} (\text{EMPREGADO})$



codCat
1
3

- Em SQL:

```
SELECT codCat FROM EMPREGADO
```



codCat
1
1
3
3

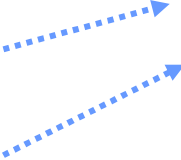
Nem sempre o resultado de um comando SQL é um conjunto !!!

## Operações Algébricas – projecção (continuação)

- Para não existirem duplicados é necessário utilizar a palavra reservada **DISTINCT**
- **Questão:** Qual a data de admissão e código da categoria de cada empregado?

- Em álgebra relacional:

–  $\pi_{\text{dataAdmissão, codCat}}(\text{EMPREGADO})$

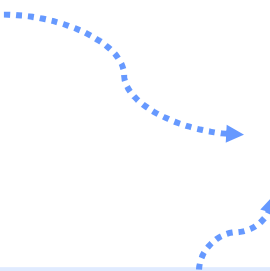


dataAdmissão	codCat
20-Mar-01	1
20-Mar-01	3
6-Abr-98	3

- Em SQL:

SELECT DISTINCT dataAdmissão, codCat FROM EMPREGADO

- Uma vez que a chave não faz parte das colunas projectadas é possível existirem duplicados



dataAdmissão	codCat
20-Mar-01	1
20-Mar-01	1
20-Mar-01	3
6-Abr-98	3

SELECT dataAdmissão, codCat FROM EMPREGADO

## Operações Algébricas – Composição de Projecção e Selecções

- **Questão:** Qual o nome e a data de admissão dos empregados com categoria igual a 3 e admitidos antes de '20-02-2000'?

nomeEmp	dataAdmissão
Carlos	6-Abr-98

- Em álgebra relacional:

–  $\pi_{\text{nomeEmp}, \text{dataAdmissão}} (\sigma_{\text{dataAdmissão} < '20-02-2000' \wedge \text{codCat}=3} (\text{EMPREGADO}))$

- Em SQL:

```
SELECT DISTINCT emp.nomeEmp, emp.dataAdmissão  
FROM (SELECT * FROM EMPREGADO  
WHERE dataAdmissão < '20-02-2000' AND codCat=3) as emp
```

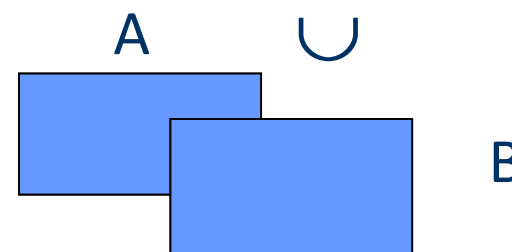
OU

```
SELECT DISTINCT nomeEmp, dataAdmissão  
FROM EMPREGADO  
WHERE dataAdmissão < '20-02-2000' AND codCat=3
```

## Operações Algébricas – União

- Na operação União, os Esquemas de Relação têm de ser compatíveis, isto é, têm que ter o mesmo grau e os atributos terem o mesmo domínio
- Considerando os Esquemas de Relação  $A(A1,A2)$ ,  $B(B1,B2)$
- Em álgebra relacional:
  - $A \cup B$
- Em SQL:

```
SELECT * FROM A
UNION
SELECT * FROM B
```



- É garantido que o resultado é um conjunto – não existem duplicados
- Se os duplicados são desejados, então usar **UNION ALL**



---

## ***Operações Algébricas – União (continuação)***

---

- **Questão:** pretende-se saber não só os nomes dos empregados do departamento com o código 2, mas também os nomes dos empregados que entraram ao serviço depois de 1998.

```
SELECT nomeEmp FROM EMPREGADO  
WHERE codDep=2  
UNION  
SELECT nomeEmp FROM EMPREGADO  
WHERE dataAdmissão>='01-01-1999'
```

OU

```
SELECT DISTINCT nomeEmp FROM EMPREGADO  
WHERE codDep=2 OR dataAdmissão>='01-01-1999'
```

---

## ***Atributo Discriminante***

---

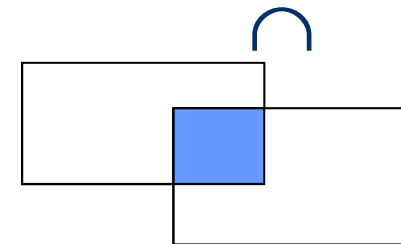
- Por vezes, é necessário efectuar um conjunto de interrogações, e ser necessário ter um atributo discriminante que identifique a origem desses dados
- Seja
  - **FUNCIONARIO(id,nome)**
  - **MOTORISTA(id, nome, nCarta)**
  - **PROFESSOR(id,nome,grau)**
- **Questão:** Quais os funcionários existentes com indicação da sua profissão

```
SELECT id, nome,'Motorista' as profissão FROM MOTORISTA
UNION
SELECT id, nome,'Professor' as profissão FROM PROFESSOR
```

## Operações Algébricas – Intersecção

- Na operação de Intersecção, os Esquemas de Relação têm de ser compatíveis, ou seja, terem o mesmo grau e os atributos terem o mesmo domínio
- Considerando os Esquemas de Relação  $A(A1,A2)$ ,  $B(B1,B2)$
- **Questão:** Quais os tuplos comuns entre A e B
- Em álgebra relacional:
  - $A \cap B$
- Em SQL:

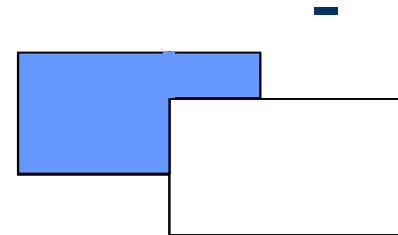
```
SELECT * FROM A  
INTERSECT  
SELECT * FROM B
```



**É garantido que o resultado é um conjunto – não existem duplicados**

## Operações Algébricas – Diferença

- Na operação de Diferença, os Esquemas de Relação têm de ser compatíveis, ou seja, terem o mesmo grau e os atributos terem o mesmo domínio
- Considerando os Esquemas de Relação  $A(A1,A2)$ ,  $B(B1,B2)$
- **Questão:** O que pertence a A mas não pertence a B
- Em álgebra relacional:
  - $A - B$
- Em SQL:



```
SELECT * FROM A
EXCEPT (MINUS para versões anteriores à SQL92)
SELECT * FROM B
```

**É garantido que o resultado é um conjunto – não existem duplicados**

---

## ***Operações Algébricas – União, Intersecção, Diferença***

---

- Nestas três operações, por omissão, não são admitidos duplicados no resultado
  - Num SELECT, pelo contrário, o comportamento por omissão é o de admitir duplicados
- É necessário ter em atenção que os SGBD fazem por vezes conversões implícitas de dados, para torná-los compatíveis
- Nesses casos é necessário consultar as tabelas de conversões e respectivas prioridades
- Tipicamente:
  - as cadeias de caracteres são convertidas na que tiver maior dimensão
  - os tipos numéricos são convertidos no que tiver maior precisão

## Operações Algébricas – Produto Cartesiano

### BANDA

Codigo	Nome	AnoFormacao	Genero
1	Metallica	1981	1
2	Madredeus	1991	2
3	Iron Maiden	1976	1
4	The Platters	1953	3

### GENERO

Codigo	Designacao
1	Metal
2	Fado
3	Rock

- **Questão:** Combinar os tuplos de BANDA com os de GENERO

- Em álgebra relacional:
  - BANDA x GENERO

- Em SQL:

```
SELECT *  
FROM BANDA, GENERO
```

Ou

```
SELECT *  
FROM BANDA CROSS JOIN GENERO
```

Codigo	Nome	AnoFormacao	Genero	Codigo	Designacao
1	Metallica	1981	1	1	Metal
1	Metallica	1981	1	2	Fado
1	Metallica	1981	1	3	Rock
2	Madredeus	1991	2	1	Metal
2	Madredeus	1991	2	2	Fado
2	Madredeus	1991	2	3	Rock
3	Iron Maiden	1976	1	1	Metal
3	Iron Maiden	1976	1	2	Fado
3	Iron Maiden	1976	1	3	Rock
4	The Platters	1953	3	1	Metal
4	The Platters	1953	3	2	Fado
4	The Platters	1953	3	3	Rock

## Operações Algébricas – Produto Cartesiano (cont.)

- Com foi visto anteriormente, o uso do Produto Cartesiano pode não ter muito interesse do ponto de vista prático
- No entanto, quando aplicada uma Selecção sobre um produto cartesiano, o interesse é óbvio
- **Questão:** Qual o género musical de cada banda

Codigo	Nome	AnoFormacao	Genero	Codigo	Designacao
1	Metallica	1981	1	1	Metal
2	Madredeus	1991	2	2	Fado
3	Iron Maiden	1976	1	1	Metal
4	The Platters	1953	3	3	Rock

- Em álgebra relacional:

–  $\sigma_{\text{genero=codigo}} (\text{BANDA} \times \text{GENERO})$

- Em SQL:

```
SELECT*  
FROM BANDA,GENERO  
WHERE BANDA.genero=GENERO.codigo
```

Esta sintaxe não é aconselhada quando o objectivo é efectuar uma junção.



## Operações Algébricas – Junção

- **Questão:** Qual o código, e designação do género musical de cada banda?
- Em álgebra relacional:

– BANDA  $\bowtie_{\text{genero=codigo}}$  GENERO

- Em SQL:

```
SELECT *  
FROM BANDA INNER JOIN GENERO  
ON (BANDA.genero=GENERO.codigo)
```

Codigo	Nome	AnoFormacao	Genero	Codigo	Designacao
1	Metallica	1981	1	1	Metal
2	Madredeus	1991	2	2	Fado
3	Iron Maiden	1976	1	1	Metal
4	The Platters	1953	3	3	Rock



---

## ***Operações Algébricas – Junção (continuação)***

---

- A operação de Join tem o formato geral:
  - **SELECT <atributos> | \***
  - **FROM <tabela1> [NATURAL] [<tipoJunção>] JOIN <tabela2>**
  - **[ON <condição>]**
- *tipoJunção* pode ter os seguintes valores:
  - INNER (⋈)
  - LEFT [OUTER] ( ⋈<sub>l</sub> )
  - RIGHT [OUTER] ( ⋈<sub>r</sub> )
  - FULL [OUTER] ( ⋈<sub>f</sub> )
- Quando omitida é considerada INNER

## ***Ambiguidade na identificação de Atributos***

- Quando são especificadas mais do que uma tabela num comando SELECT, por vezes existe ambiguidade da identificação dos atributos
- Seja
  - Empregado(codigo, nome, codigoDepart)
  - Departamento(codigo, nome)
- Pretende-se saber o código do empregado e o nome do departamento onde ele trabalha:
  - $\pi_{1,5} (\text{Empregado} \bowtie_{3=1} \text{Departamento})$

```
SELECT codigo, nome  
FROM Empregado INNER JOIN Departamento  
On (codigoDepart=codigo)
```

Será que a implementação em SQL está correcta?

## ***Ambiguidade na identificação de Atributos (continuação)***

- A resposta é não!!
- Existem várias ambiguidades:
  - *Codigo* é um atributo de que tabela?
  - *Nome* é um atributo de que tabela?
  - Quando se compara *codigo* a *codigoDepart*, *codigo* é um atributo de que tabela?
- Correctamente

```
SELECT Empregado.codigo, Departamento.nome  
FROM Empregado INNER JOIN Departamento  
On (Empregado.codigoDepart = Departamento.codigo)
```

**A ambiguidade é resolvida precedendo o nome do atributo pelo nome da tabela, separados por um ponto !**

## Ambiguidade na identificação de Atributos (continuação)

- Mesmo quando apenas uma única tabela está envolvida num SELECT podem existir ambiguidades
- Considere-se o seguinte Esquema Relacional
  - **EMPREGADO**(numBI, primNome, ultNome, numBIChefe )
- Questão: Para cada empregado, pretende-se saber o seu primeiro e último nome e também o primeiro e último nome do seu chefe

$\pi_{2,3,6,7} (\text{Empregado} \bowtie_{4=1} \text{Empregado})$

```
SELECT Emp.primNome, Emp.ultNome,  
       Chefe.primNome, Chefe.ultNome  
FROM Empregado as Emp INNER JOIN Empregado as Chefe  
ON(Emp.numBIChefe = Chefe.numBI)
```

## ***Combinação de operações – Junção, projecção, Selecção***

- Por vezes, a interrogação que se pretende leva a que sejam combinadas um conjunto de operações
- Seja
  - **Material(nome, codigoMaterial)**
  - **Fornece(codigoFornecedor,codigoMaterial)**
- **Questão:** Qual o nome dos materiais fornecidos pelo fornecedor de código 123?
- Em Álgebra Relacional:
  - $\pi_{[1]} (\sigma_{[3]=123} (\text{Material} \bowtie_{[2]=[2]} \text{Fornece}))$
- Em SQL

```
SELECT Material.nome
FROM Fornecedor INNER JOIN Material
ON(Fornece.codigoMaterial=Material.codigoMaterial)
WHERE Fornecedor.codigoFornecedor=123
```

## Combinação de operações – Junção, projecção, Selecção (cont.)

- Ou ainda, em Álgebra Relacional

$$- \pi_{[1]} (\text{Material} \bowtie_{[2]=[2]} (\sigma_{[1]=123} (\text{Fornece})))$$

- Em Sql

```
SELECT Material.nome
FROM Material INNER JOIN
    (SELECT * FROM Fornece
     WHERE Fornecedor.codigoFornecedor=123) as F
ON(F.codigoMaterial=Material.codigoMaterial)
```

Nesta resolução, os tuplos usados na junção são minimizados, pois a selecção é efectuada antes da junção!

---

## *Operadores de comparação*

---

- Quando na construção de um junção, não é necessário que o operador de comparação usado seja o operador '='
- Podem ser usados quaisquer operadores de comparação e predicados (referidos mais adiante)
- Operadores de comparação – '=', '>', '<', '>=', '<=', '<>'
- Seja
  - Aluno(num,nota,disciplina)
  - Aprovado(notaMax,notaMin,classificacao)
- Qual a classificação para cada disciplina do aluno 123?

```
SELECT A1.disciplina,Ap.classificacao  
FROM Aprovado as Ap INNER JOIN Aluno as A1  
ON(A1.nota >=Ap.notaMin AND A1.nota <notaMax)  
WHERE A1.num=123
```

## ***Junções encadeadas***

- Seja
  - **Material**(codigoMaterial ,nome)
  - **Fornecedor**(codigoFornecedor, nome)
  - **Fornece**(codigoFornecedor, codigoMaterial)
- Qual o nome do fornecedor e os nomes dos materiais por ele fornecido?

```
SELECT Fornecedor.nome,Material.nome  
FROM Fornecedor INNER JOIN Fornece  
ON(Fornecedor.codigoFornecedor = Fornece.codigoFornecedor)  
INNER JOIN Material ON(Fornece.codigoMaterial =  
Material.codigoMaterial)
```

Ou

```
SELECT Fornecedor.nome,Material.nome  
FROM Fornecedor NATURAL JOIN Fornece NATURAL JOIN Material
```

Não é aconselhado o seu uso, devido aos erros que pode provocar.





---

## ***Junções encadeadas - continuação***

---

- Ou ainda

```
SELECT Fornecedor.nome, Material.nome
FROM Fornecedor, Fornece, Material
WHERE Fornece.codigoMaterial = Material.codigoMaterial
AND Fornecedor.codigoFornecedor =
                                Fornece.codigoFornecedor
```

Todas estas versões apenas diferem na forma como são escritas, ou seja, na sintaxe.

O interpretador poderá gerar o mesmo plano de execução para as três versões !!

## ***Operações Algébricas – Junção Externa***

- **Questão:** Pretende-se listar todos os fornecedores e todos os produtos, mostrando-se quem fornece cada um

- Em Álgebra Relacional

– Material  $\bowtie_{[1]=[2]}$  Fornece  $\bowtie_{[1]=[2]}$  Fornecedor

- Em Sql

```
SELECT Material.*,Fornecedor.*  
FROM Material FULL OUTER JOIN Fornece  
ON(Fornece.codigoMaterial =  
    Material.codigoMaterial)  
FULL OUTER JOIN Fornecedor  
ON(Fornecedor.codigofornecedor =  
    Fornece.codigofornecedor)
```

## Operações Algébricas – Junção Externa (cont.)

- Seja
  - EMPREGADO( codEmp, nome, codCategoria )
  - CATEGORIA( codCategoria, designacao, ordenado )
- **Questão:** Quais os empregados e categorias existentes e para cada empregado qual as categorias superiores à sua?

- Em Álgebra Relacional
  - Empregado  $\bowtie_{[3]<[1]}$  Categoria

- Em SQL

```
SELECT Empregado.*,Categoria.*  
FROM Empregado FULL JOIN Categoria  
ON(Empregado.codCategoria < Categoria.codCategoria)
```

Poderá ser feita de outra forma?


## Operações Algébricas – Junção Externa (à Esquerda e Direita)

- Sim!
- Em Álgebra Relacional
  - $(\text{Empregado} \bowtie_{[3]<[1]} \text{Categoria}) \cup (\text{Empregado} \bowtie_{[3]<[1]} \text{Categoria})$


- Em SQL

```
SELECT Empregado.*,Categoria.*
FROM Empregado LEFT JOIN Categoria
ON(Empregado. codCategoria < Categoria.codCategoria)
UNION
SELECT Empregado.*,Categoria.*
FROM Empregado RIGHT JOIN Categoria
ON(Empregado. codCategoria < Categoria.codCategoria)
```

## Operações Algébricas – Junção Externa à Esquerda

- **Questão:** Quais os empregados e as categorias inferiores á sua?
- Em Álgebra Relacional
  - Empregado  Categoria
- Em SQL

```
SELECT Empregado.*,Categoria.*  
FROM Empregado LEFT JOIN Categoria  
ON(Empregado. codCategoria > Categoria.codCategoria)
```

- **Questão:** Quais categorias existentes e seus empregados ?
- Em Álgebra Relacional
  - Empregado  Categoria
- Em SQL

```
SELECT Categoria.*, Empregado.*  
FROM Empregado RIGHT JOIN Categoria  
ON(Empregado.codCategoria = Categoria.codCategoria )
```

---

## ***Funções de agregação***

---

- **Questão:** Quantos empregados existem na empresa?
- Não é possível responder a esta questão com o que foi apresentado até aqui!
- Existem um conjunto de funções que efectuam operações sobre conjunto de linhas
- Nomeadamente:
  - COUNT – conta o número de linhas
  - SUM – efectua o somatório de valores
  - AVG – encontra a média de valores
  - MAX – determina o maior valor
  - MIN – determina o menor valor
- Estas funções designam-se funções de agregação

---

## ***Funções de agregação (cont.)***

---

- COUNT( \* )
  - Conta o número total de linhas (incluindo os valores NULL)
- COUNT( [DISTINCT] | [ALL] <coluna> )
  - Conta o número de linhas excluindo as que, para a coluna indicada, têm valor NULL. Caso se use DISTINCT, não se consideram valores duplicados. Se apenas for indicado o nome da coluna, por omissão é considerado o ALL (os duplicados são incluídos)
- Para responder à questão:
  - $\int$  Count(codEmpregado) (Empregado)

**SELECT COUNT(codEmpregado) FROM Empregado**

- De notar que não é necessário utilizar o DISTINCT, pois a coluna sobre a qual é feita a contagem é chave primária da tabela, ou seja, não admite valores iguais nem NULLs

---

## ***Funções de agregação (cont.)***

---

- SUM([DISTINCT] | [ALL] <expressão escalar>)
  - Efectua o somatório dos valores da expressão. Quando especificado DISTINCT, apenas os valores diferentes são tidos em conta. Por omissão os duplicados são incluídos (ALL)
- AVG([DISTINCT] | [ALL] <expressão escalar>)
  - Efectua a média dos valores da expressão. Quando especificado DISTINCT, apenas os valores diferentes são tidos em conta. Por omissão os duplicados são incluídos (ALL). É equivalente a SUM/COUNT
- MAX(<expressão escalar>)
  - Determina o máximo valor na expressão.
- MIN(<expressão escalar>)
  - Determina o mínimo valor na expressão.



---

## ***Funções de agregação (cont.)***

---

- Estas funções de agregação apenas podem aparecer na cláusula SELECT ou na cláusula HAVING (apresentada mais à frente)
- O argumento (expressão escalar) das funções SUM e AVG tem de ser numérico
- A expressão escalar não pode ser ela própria um resultado de uma função de agregação:
  - SELECT AVG ( SUM (QTY) ) as QT FROM .....
  - A cláusula acima é ilegal!
- Se o resultado da expressão escalar é vazio, o resultado da função COUNT é zero, o das outras é NULL
- Quando no SELECT aparecem misturados atributos resultantes de funções agregadoras com outros, essa expressão é ilegal!

---

## **GROUP BY**

---

- A seguinte expressão é ilegal:
  - SELECT Atr1, AVG (Atr2) FROM TAB1
  - O resultado de uma selecção de um atributo sobre uma tabela, possivelmente terá vários valores
  - O resultado de uma função agregadora é, sempre, um único valor!
- É necessário utilizar um mecanismo de agrupamento, da mesma forma que foi feito na álgebra relacional
- Em Álgebra Relacional, existe o operador de agrupamento que aplicava as funções agregadores sobre agrupamentos de valores
- Em SQL, o agrupamento é efectuado através da cláusula **GROUP BY**

---

## ***GROUP BY (cont.)***

---

- A cláusula GROUP BY tem a forma:
  - GROUP BY <lista colunas>
  - Onde <lista de colunas> é uma lista de colunas separadas por virgula, sobre as quais será feito o agrupamento
- **Questão:** qual o maior dos ordenados, para cada departamento?

- Em Álgebra Relacional

`codigo`  $\bowtie$  `Max(ordenado)` (DEPARTAMENTO)

- Em SQL

```
SELECT codigo as Departamento , MAX(ordenado) as
      MaiorOrdenado
FROM DEPARTAMENTO
GROUP BY codigo
```

---

## ***GROUP BY (cont.)***

---

- Quando é especificada a cláusula GROUP BY, na cláusula SELECT apenas podem aparecer:
  - As colunas que são especificadas na cláusula GROUP BY
  - Resultados de funções agregadores
- Qualquer outra coluna especificada na cláusula SELECT dá origem a instruções ilegais:

```
SELECT TABELA.*, COUNT(*)  
FROM TABELA
```

```
SELECT TABELA.CODIGO, SUM(CODIGO)  
FROM TABELA
```

---

## HAVING

---

- A cláusula WHERE é verificada para cada linha da tabela, ficando esse linha no resultado final se verificar a condição
- Por vezes, apenas se querem obter resultados sobre grupos quando estes verificam uma determinada condição
  - Com a cláusula WHERE não se consegue isso
- Seja: **FUNCIONARIO(cod, nome, ordenado, codDepartamento)**
- **Questão:** Quais os códigos dos departamentos e o maior dos salários, onde a média seja maior que 1000€?
- É necessário aplicar uma condição a cada grupo de funcionários que pertençam ao mesmo departamento; só os grupos que verificarem a condição são considerados
- Essa condição é indicada usando a cláusula HAVING

---

## ***HAVING (cont.)***

---

- A cláusula HAVING tem a forma:
  - HAVING <condição>
  - Onde <condição> é uma expressão cujo resultado é *booleano*, com um único valor por grupo
- Para responder à questão colocada:

```
SELECT codDepartamento, MAX(Ordenado)
FROM FUNCIONARIO
GROUP BY codDepartamento
HAVING AVG(ordenado) > 1000
```
- Uma diferença entre as cláusulas HAVING e WHERE:
  - A cláusula HAVING deve sempre conter funções de agregação
  - A cláusula WHERE nunca contém, directamente, funções de agregação

---

## ***ORDER BY***

---

- Por vezes deseja-se que o resultado de uma interrogação venha ordenado por um determinado critério
- Essa ordenação é feita utilizando a cláusula ORDER BY
- A cláusula ORDER BY tem a forma:
  - ORDER BY <coluna | número da coluna [ASC|DESC] >
  - coluna indica a coluna sobre qual a ordenação vai ser feita
  - número da coluna indica qual a coluna (posicionalmente) sobre a qual a ordenação irá ser efectuada. Esse número de ordem nada tem a ver com as colunas existentes numa tabela, mas sobre o número de ordem que essa coluna tem na cláusula SELECT
  - ASC indica que se pretende uma ordenação ascendente
  - DESC indica que se pretende uma ordenação descendente
- Por omissão, o tipo de ordenação é ascendente

---

## ***ORDER BY (cont.)***

---

- **Questão:** Qual o nome e o departamento dos funcionários existentes, ordenados alfabeticamente?

```
SELECT nome, codDepartamento as Departamento  
FROM FUNCIONARIO  
ORDER BY nome ASC
```

- Quais os códigos dos departamentos e o maior dos salários, onde a média seja maior que 1000€, ordenados por ordem decrescente de salários?

```
SELECT codDepartamento, MAX(Ordenado)  
FROM FUNCIONARIO  
GROUP BY codDepartamento  
HAVING AVG(ordenado)>1000  
ORDER BY 2 DESC
```



---

## ***SELECT – A sintaxe com GROUP BY, HAVING e ORDER BY***

---

- A sintaxe do SELECT, com a inclusão das cláusulas GROUP BY, HAVING e ORDER BY fica então:

```
SELECT [DISTINCT] <colunas> | *  
FROM <lista tabelas>  
[WHERE <condição>]  
[GROUP BY <lista colunas> ]  
[HAVING <condição>]  
[ORDER BY <coluna | número da Coluna [ASC|DESC] > ]
```

---

## ***Sub-Interrogação***

---

- Seja:
  - CATEGORIA( codCat, nome, salarioBase )
  - DEPARTAMENTO( codDep, nome, localizacao )
  - EMPREGADO( codEmp, nome, salarioEfectivo, codCat, codDep )
- **Questão:** Qual o nome dos empregados que trabalham no mesmo departamento que o(s) empregado(s) com nome 'João Maria' ?
- **Solução:**

```
SELECT nome
FROM EMPREGADO as EP1 INNER JOIN EMPREGADO as EP2
ON(EP1.codDep=EP2.codDep)
WHERE EP1.nome<>EP2.nome AND EP1.nome='João Maria'
```
- Existe no entanto outra solução possível – Separar a interrogação em duas partes

---

## ***Sub-Interrogação (cont.)***

---

- É necessário responder então a duas sub-questões :
  - Qual o código do departamento do(s) empregado(s) 'João Maria'?

```
SELECT DISTINCT E2.codDep  
FROM EMPREGADO AS E2  
WHERE E2.nome = 'João Maria'
```

- Qual o nome dos empregados do(s) departamento(s) com o(s) código(s) obtido(s) na interrogação anterior, mas que não são 'João Maria' ?
  - assumindo que existem cinco empregados com o nome 'João Maria' e que três deles estão no departamento 4, um está no 7 e outro no 11

```
SELECT DISTINCT E1.nome  
FROM EMPREGADO AS E1  
WHERE E1.nome <> 'João Maria' AND E1.codDep IN (4,7,11)
```

---

## ***Sub-Interrogação (cont.)***

---

- Para responder à questão inicial:

```
SELECT DISTINCT E1.nome
FROM EMPREGADO AS E1
WHERE E1.nome <> 'João Maria' AND E1.codDep IN
    ( SELECT DISTINCT E2.codDep
      FROM EMPREGADO AS E2
      WHERE E2.nome = 'João Maria'
    )
```

- Foi utilizada uma sub interrogação (SELECT interior) para responder à questão
- Foi utilizado o predicado IN (abordado mais adiante) para verificar se um valor pertence ao conjunto

---

## ***Sub-Interrogação não correlacionada***

---

- Numa sub-interrogação **não** correlacionada, a interrogação interior não necessita de valores da interrogação exterior. Era o caso do exemplo anterior
- **Questão:** Quais os códigos e nomes das categorias com menor salário base?

```
SELECT CO.codCat, CO.nome FROM CATEGORIA AS CO  
WHERE CO.salarioBase=( SELECT MIN( CI.salarioBase)  
                        FROM CATEGORIA AS CI )
```

- A interrogação interior (SELECT MIN... ) não depende da exterior
  - a interrogação interior é executada em primeiro lugar e apenas uma vez
  - a relação devolvida na interrogação interior permite resolver a exterior

---

## ***Sub-Interrogação correlacionada***

---

- Numa sub-interrogação correlacionada a interrogação interior necessita de valores da interrogação exterior
- **Questão:** Quais as categorias cujo salário base é inferior a metade do valor médio dos salários efectivos dos empregados dessas categorias?

```
SELECT C.* FROM CATEGORIA AS C
WHERE C.salarioBase < (
    (SELECT AVG(E.salarioEfectivo)
     FROM EMPREGADO As E
     WHERE E.codCat=c.codCat
    )/2)
```

- A interrogação interior (SELECT AVG... ) depende da exterior
  - a informação da interrogação exterior é passada à interior
  - para cada linha da interrogação exterior é executada a interior

---

## ***Predicados***

---

- Quando foi introduzida a sintaxe geral de uma interrogação SQL, uma das cláusulas existentes era o WHERE
- Esta cláusula foi apresentada no contexto de um SELECT, tendo o formato
  - [WHERE <condição>]
- Onde
  - <condição> é uma expressão lógica que define a condição a verificar
- Essa condição pode ser
  - Um conjunto de comparações combinadas entre si, e/ou
  - Uma colecção de Predicados combinados entre si
  - A combinação é feita recorrendo aos operadores lógicos AND, OR e NOT
- Cada Predicado quando avaliado produz um valor lógico verdadeiro ou falso

---

## ***Predicados (cont.)***

---

- Os Predicados podem ser utilizados num contexto estático, sendo avaliados com base em valores constantes.
  - ...WHERE E1.codDep IN ( 4, 7, 11 ) ...
- Podem, no entanto, ser usados com base em valores dinâmicos, a retirar da base de dados
  - ...WHERE E1.codDep IN ( SELECT E2.codDep FROM EMPREGADO )...
- Alguns dos predicados existentes são:
  - de Comparação
    - **BETWEEN** - *WHERE ATR1 BETWEEN 1 AND 5*
    - **LIKE** - *WHERE ATR1 LIKE '%123'*
    - **IN** – *WHERE ATR1 IN (1,2,3,10)*
    - **ALL, ANY** - *WHERE Atr1 > ANY( SELECT ... )*
    - **EXISTS** - *WHERE EXISTS( SELECT ... )*
  - Teste de valor nulo
    - **IS NULL** - *WHERE Atr1 IS NULL*



---

## ***Predicados - BETWEEN***

---

- O predicado BETWEEN não é mais que uma forma simplificada de escrever algumas condições
- A sintaxe do BETWEEN
  - <construtor de linha> [NOT] BETWEEN <construtor de linha> AND <construtor de linha>
- Semanticamente:
  - $Y \text{ BETWEEN } X \text{ AND } Z$  é equivalente a
  - $X \leq Y \text{ AND } Y \leq Z$
- **Questão:** Qual o nome e o código dos empregados que têm o salário efectivo entre 1000€ e 2000€?

```
SELECT nome, codEmp  
FROM EMPREGADO  
WHERE salarioEfectivo BETWEEN 1000 AND 2000
```

---

## ***Construtor de linha***

---

- Um construtor de linha é usado no predicado BETWEEN e noutros abordados de seguida
- Um construtor de linha pode ser:
  - um átomo (por ex. uma coluna )
  - uma expressão que origina uma tabela, entre parêntesis curvos. Nesse caso, o resultado dessa expressão deve ser uma tabela com pelo menos uma linha
- Se forem efectuadas comparações entre construtores de linha que sejam avaliados em tabelas, estas têm de ter o mesmo grau

---

## ***Construtor de linha (cont.)***

---

- A comparação faz-se linha a linha e coluna a coluna
- Seja E e D os construtores de linha Esquerdo e direito e N o seu grau
  - $E = D$  é verdade se  $E_i = D_i$ , para cada linha e para todo o  $i, i \in \{1 \dots N\}$
  - $E < D$  é verdade se  $E_i < D_i$ , para cada linha e para todo o  $i, i \in \{1 \dots N\}$
- Este raciocínio aplica-se aos restantes operadores de comparação

---

## ***Predicados – LIKE***

---

- O predicado LIKE é usado para encontrar padrões em cadeias de caracteres, cuja sintaxe é
  - <expressão> [NOT] LIKE <padrão> [ESCAPE <caracterExcepção>]
- <expressão> tem de ser uma cadeia de caracteres
- <padrão> é constituído pelo padrão que se quer encontrar na cadeia de caracteres, podendo incluir meta-caracteres, que não sendo precedidos do carácter de excepção, têm o seguinte significado:
  - O símbolo '%', quando a iniciar ou a terminar o padrão, indica qualquer sequência de caracteres
  - O símbolo '\_' quando a iniciar ou a terminar o padrão, indica um carácter qualquer. Pode ser usado várias vezes
  - caracterExcepção ocorre em padrão anulando o tratamento especial

---

## ***Predicados - LIKE (cont.)***

---

- **Questão:** Qual o nome e código dos empregados que tem 'João' no nome?

```
SELECT nome, codEmp  
FROM EMPREGADO  
WHERE nome LIKE '%João%'
```

- **Questão:** Qual o nome e código dos empregados cujo nome começa por 'João'?

```
SELECT nome, codEmp  
FROM EMPREGADO  
WHERE nome LIKE 'João%'
```

---

## ***Predicados – IN***

---

- O predicado IN é usado para verificar se um determinado valor está contido numa determinada lista de valores
- A sintaxe do IN
  - <construtor de linha> [NOT] IN (<sub-interrogação> | <lista de expressões escalares>)

- **Questão:** Qual o nome e código dos empregados que pertencem aos departamentos 2 e 3?

```
SELECT nome,codEmp FROM EMPREGADO AS E1  
WHERE E1.codDep IN (2,3)
```

- **Questão:** Quais as categorias dos empregados com maior salário efectivo?

```
SELECT C.* FROM CATEGORIA AS C WHERE C.codCat IN  
( SELECT E.codCat FROM EMPREGADO AS E WHERE  
  E.salarioEfectivo = ( SELECT MAX( E.salarioEfectivo )  
                      FROM EMPREGADO AS E ) )
```

---

## ***Predicados – ANY, ALL***

---

- Estes predicados verificam se alguma ou todas as linhas têm um atributo que obedece a uma expressão envolvendo operadores relacionais
- A sintaxe do ANY,ALL
  - <construtor de linha> <operador de comparação> ANY | ALL (sub-interrogação)
- **Questão:** Qual o nome dos empregados cujo salário efectivo é superior ao de alguns empregados da mesma categoria ?

```
SELECT E.nome
FROM EMPREGADO AS E
WHERE E.salarioEfectivo > ANY
      ( SELECT E1.salarioEfectivo
        FROM EMPREGADO E1
        WHERE E.codCat = E1.codCat )
```

## ***Predicados – ANY, ALL (cont.)***

- **Questão:** Qual o nome dos empregados cujo salário efectivo é superior ao de todos os empregados de departamentos localizados em 'Lisboa' ?

```
SELECT E.nome FROM EMPREGADO AS E
WHERE E.salarioEfectivo > ALL
      ( SELECT E1.salarioEfectivo
        FROM EMPREGADO E1 INNER JOIN DEPARTAMENTO D
        ON (E1.codDep = D.codDep )
        WHERE D.localização='Lisboa' )
```

○ Ou

```
SELECT E.nome FROM EMPREGADO AS E
WHERE E.salarioEfectivo >
      ( SELECT MAX(E1.salarioEfectivo)
        FROM EMPREGADO E1 INNER JOIN DEPARTAMENTO D
        ON (E1.codDep = D.codDep )
        WHERE D.localização='Lisboa' )
```



---

## ***Predicados – EXISTS***

---

- Este predicado é utilizado para testar se uma determinada tabela tem pelo menos uma linha
- A sintaxe do EXISTS
  - [NOT] EXISTS (sub-interrogação)
- **Questão:** Quais os departamentos que têm pelo menos um empregado?

```
SELECT D.*  
FROM Departamento as D  
WHERE EXISTS (SELECT E.codEmp FROM EMPREGADO as E  
              WHERE E.codDep = D.codDep)
```

- Se o resultado da sub-interrogação não for vazio, o predicado EXISTS retorna *true*

---

## ***Predicados – EXISTS (cont.)***

---

- A Questão anterior pode escrever-se em álgebra relacional
  - $\text{DEPARTAMENTO} \cap (\pi_{1,2,3}(\text{DEPARTAMENTO} \bowtie_{1=5} \text{EMPREGADO}))$
- Então, uma das utilizações que se pode dar ao predicado EXISTS é a realização do operador INTERSECT

```
SELECT D.*  
FROM DEPARTAMENTO AS D  
WHERE EXISTS  
  ( SELECT *  
    FROM DEPARTAMENTO AS D1  
      INNER JOIN EMPREGADO AS E  
        ON ( D1.codDep = E.codDep )  
    WHERE D.codDep = D1.codDep )
```

---

## ***Predicados – EXISTS (cont.)***

---

- Para responder à mesma questão sem recorrer ao EXISTS

```
SELECT D.* FROM DEPARTAMENTO AS D
WHERE 1<=(SELECT COUNT(E.codEmp)
          FROM EMPREGADO
          WHERE E.codDep = D.codDep )
```

- **Questão:** Qual o código e nome das categorias que não têm empregados ?

```
SELECT D.*
FROM Departamento AS D
WHERE NOT EXISTS (SELECT E.codEmp
                  FROM EMPREGADO AS E
                  WHERE E.codDep=D.codDep)
```

## Operações Algébricas – Divisão

- Para as seguintes tabelas

Empregado

codEmp	nomeEmp	codCat	codDept
1	António	1	1
2	João	1	2
3	Nuno	3	3
4	Carlos	2	2
5	Carlos	3	2

Categoria

codCat	designacao	salarioBase
1	CategoriaA	1.500,00 €
2	CategoriaB	1.100,00 €
3	CategoriaC	750,00 €

- Questão:** Quais os códigos dos departamentos que têm empregados de todas as categorias ?

$$- \pi_{3,4} ( \text{EMPREGADO} ) \div \pi_1 ( \text{CATEGORIA} ) \quad \dots\dots\dots \rightarrow \begin{array}{|c|} \hline \text{codDept} \\ \hline 2 \\ \hline \end{array}$$

Qual será a resolução desta questão, utilizando a interrogação SQL?

---

## ***Operações Algébricas – Divisão (cont.)***

---

- Colocando a questão de outra forma:
  - Quais os códigos dos departamentos para os quais, qualquer que seja a categoria, existe algum empregado desse departamento e dessa categoria
- Utilizando uma notação simbólica:  
**codigoDepartamento :  $\forall$  categoria  $\in$  CATEGORIA**  
**(  $\exists$  empregado :**  
EMPREGADO.codDep = codigoDepartamento AND  
EMPREGADO.codCat = CATEGORIA.codCat )
- Designando por *p(empregado)* o que está entre parêntesis

---

## ***Operações Algébricas – Divisão (cont.)***

---

- Sabendo que,  $\forall x : p(x) \Leftrightarrow \neg \exists x : \neg p(x)$
- Tem-se  
    **codigoDepartamento :  $\neg \exists \text{ categoria} \in \text{CATEGORIA},$   
    (  $\neg \exists p(\text{codigoDepartamento})$  )**
- Ou seja  
    **codigoDepartamento :  $\neg \exists \text{ categoria} \in \text{CATEGORIA},$   
    (  $\neg \exists \text{ empregado} :$   
    **EMPREGADO.codDep = codigoDepartamento**  
    **AND EMPREGADO.codCat = CATEGORIA.codCat )****

## Operações Algébricas – Divisão (cont.)

1. `codigoDepartamento` :
2.  $\neg \exists$  categoria  $\in$  CATEGORIA,
3.  $( \neg \exists$  empregado : EMPREGADO.codDep = `codigoDepartamento`  
AND EMPREGADO.codCat = CATEGORIA.codCat )

- Em SQL

```
SELECT D.codigoDepartamento
FROM ( SELECT DISTINCT codDep FROM EMPREGADO )
      AS D(codigoDepartamento)
WHERE NOT EXISTS
      ( SELECT * FROM CATEGORIA AS C
        WHERE NOT EXISTS
          ( SELECT * FROM EMPREGADO AS E
            WHERE E.codDep = D.codigoDepartamento
                  AND E.codCat = C.codCat ) )
```

Diagram illustrating the SQL query structure with annotations:

- 1: Points to the subquery `( SELECT DISTINCT codDep FROM EMPREGADO ) AS D(codigoDepartamento)`.
- 2: Points to the subquery `( SELECT * FROM CATEGORIA AS C WHERE NOT EXISTS ( SELECT * FROM EMPREGADO AS E WHERE E.codDep = D.codigoDepartamento AND E.codCat = C.codCat ) )`.
- 3: Points to the `WHERE NOT EXISTS` clause.

## ***Operações Algébricas – Divisão (cont.)***

- Pretende-se a informação completa sobre os departamentos que têm empregados de todas as categorias ?
- Em Álgebra Relacional
  - $\text{DEPARTAMENTO} \bowtie_{1=1} (\pi_{3,4} (\text{EMPREGADO}) \div \pi_1 (\text{CATEGORIA}))$
- Em SQL (não utilizando a junção)

```
SELECT D.* FROM DEPARTAMENTO AS D
WHERE NOT EXISTS
    ( SELECT * FROM CATEGORIA AS C
      WHERE NOT EXISTS
        ( SELECT * FROM EMPREGADO AS E
          WHERE E.codDep = D.codDep AND
                E.codCat = C.codCat ) )
```



---

## *Valores NULL*

---

- Quando um determinado valor é desconhecido ou indefinido, existe um valor especial para representar isso – o NULL
- Algumas situações onde o NULL é aplicado:
  - o atributo não é aplicável para determinado tuplo
  - o atributo tem um valor desconhecido para determinado tuplo
  - o atributo tem valor conhecido mas o valor está ausente nesse instante, ou seja, ainda não foi registado na Base de Dados
  - pode ser o valor por omissão para uma determinada coluna
- Algumas características
  - É independente do domínio - inteiro, real, carácter, data, etc
  - Uma expressão com um operador de comparação será avaliada como FALSE, se algum dos seus operandos tiver o valor NULL
  - Existem funções para determinar se o valor é NULL e alterá-lo

---

## ***Manipulação de NULL***

---

- O predicado IS NULL permite determinar se um valor é NULL
- A sintaxe do IS NULL
  - <construtor linha> IS [NOT] NULL
- **Questão:** Quais os clientes que têm telefone? (admitindo que esse atributo é opcional)  

```
SELECT * FROM CLIENTE WHERE telefone IS NOT NULL
```
- A função NULLIF(X,Y) devolve NULL se X e Y forem iguais. Caso contrário devolve X
- A função COALESCE(X,Y) devolve X se este for diferente de NULL, devolvendo Y caso contrário
  - Se quisermos listar os clientes sem apresentar NULL

```
SELECT cliente.nome, COALESCE(cliente.telefone, 'Não disponível') FROM CLIENTE
```

---

## ***Tipos de dados do standard SQL2 (ANSI SQL)***

---

- **CHARACTER(n)** – cadeia de caracteres de dimensão n, fixa,  $n > 0$ 
  - Utiliza-se CHAR como abreviação
  - Existe a variante **NCHAR(n)**, que inclui suporte a caracteres *unicode*
- **CHARACTER VARYING(n)** – cadeia de caracteres de dimensão variável, com um máximo de n caracteres,  $n > 0$ 
  - Utiliza-se VARCHAR como abreviação
  - Existe a variante NVARCHAR(n) , que inclui suporte a caracteres *unicode*
- **BIT(n)** – cadeia de bits com dimensão fixa de n bits,  $n > 0$
- **BIT VARYING(n)** – cadeia de bits com dimensão variável com um máximo de n bits,  $n > 0$
- **NUMERIC(p,q)** – número decimal com p dígitos e sinal, com q casas decimais, a contar da direita,  $0 \leq q \leq p$ ,  $p > 0$ 
  - NUMERIC(p) é uma abreviação de NUMERIC(p,0)
  - a precisão do número é exactamente de p dígitos

---

## ***Tipos de dados do standard SQL2 (ANSI SQL) (cont.)***

---

- **DECIMAL(p,q)** – número decimal com  $p$  dígitos e sinal, com  $q$  casas decimais, a contar da direita,  $0 \leq q \leq p \leq m$ ,  $p > 0$ 
  - DEC é uma abreviação de DECIMAL
  - DECIMAL(p) é uma abreviação de DECIMAL(p,0)
  - A precisão do número pode não ser  $p$ , podendo ter uma precisão  $m$  maior
- **INTEGER** – número inteiro, com sinal, decimal ou binário
  - INT é uma abreviação de INTEGER
- **SMALLINT** – número inteiro, com sinal, decimal ou binário
  - SMALLINT terá sempre uma precisão nunca superior a INT
- **FLOAT(p)** – número de virgula flutuante
  - FLOAT é uma abreviação de FLOAT(p), onde  $p$  depende da implementação
  - REAL é uma alternativa a FLOAT(s), onde  $s$  depende da implementação
  - DOUBLE PRECISION é uma alternativa a FLOAT(d), onde  $d$  depende da implementação

---

## ***Caso prático - Alguns Tipos de dados no SQL Server***

---

- Binários: BINARY[(n)], VARBINARY[(n | **MAX**)]
- Carácter: CHAR[(n)], VARCHAR[(n)]
- Caracteres Unicode : NCHAR[(n)], NVARCHAR[(n | **MAX**)]
- Data e Hora: DATE, TIME, DATETIME, SMALLDATETIME
- Numérico exacto: DECIMAL[(p[,s])], NUMERIC[(p[,s])]
- Numérico aproximado: FLOAT[(n)], REAL
- Inteiro: BIGINT, INT, SMALLINT, TINYINT
- Monetário: MONEY, SMALLMONEY
- Outros: XML, TIMESTAMP, UNIQUEIDENTIFIER, BIT

---

## ***Caso prático - Alguns Tipos de dados no SQL Server (cont.)***

---

- Para os tipos **char** e **binary**, o valor de **n** pode variar entre 1 e 8000. O valor é medido em bytes.
- Para os tipos **varchar** e **varbinary**, para além do **n** pode tomar valores entre 1 e 8000, é também possível especificar a palavra reservada **MAX**, sempre que a dimensão esperada exceda os 8K. Neste caso a dimensão máxima pode ir até aos  $2^{31}-1$  bytes ( $\approx 2\text{GB}$ )
- Para aos tipos **nchar** e **nvarchar**, o valor de **n** pode variar entre 1 e 4000 bytes, tendo **nvarchar(MAX)** a mesma capacidade de armazenamento que **varchar(MAX)**.
- Os tipos de dimensão variável, podem armazenar sequencias com dimensão nula, mas ocupam sempre 2 bytes extra

---

## ***Caso prático - Alguns Tipos de dados no SQL Server (cont.)***

---

- O tipo de dados **DATETIME** permite armazenar datas desde 1 de Janeiro de 1753 até 31 Dezembro 9999. Tem precisão de 3.33 milissegundos
- O tipo de dados **SMALLDATETIME** permite armazenar datas desde 1 de Janeiro de 1900 até 6 de Junho de 2079, com precisão ao minuto
- O tipo **DATE** permite armazenar datas desde 1 de Janeiro de 0001 até 31 Dezembro 9999, com precisão ao dia
- O tipo **TIME** permite armazenar informação temporal, na gama [00:00:00.0000000 até 23:59:59.9999999]. A precisão pode ser passada como argumento. Por exemplo, TIME(7) tem precisão até 100ns. É esse o valor por omissão.

---

## ***Caso prático - Alguns Tipos de dados no SQL Server (cont.)***

---

- Os tipos **DECIMAL** e **NUMERIC** são equivalentes, sendo suportados os dois por questões de compatibilidade
  - A precisão  $p$  máxima é 38
  - Quando é utilizada a precisão máxima, podem representar-se números de  $-10^{38} + 1$  até  $10^{38} + 1$
  - Para  $1 \leq p \leq 9$  são necessários 5 bytes para armazenar os dados
  - Para  $10 \leq p \leq 19$  são necessários 9 bytes para armazenar os dados
  - Para  $29 \leq p \leq 38$  são necessários 17 bytes para armazenar os dados
- O tipo **BIGINT** utiliza 8 bytes para armazenar inteiros compreendidos entre  $-2^{63}$  e  $2^{63} - 1$
- O tipo **INT** utiliza 4 bytes (números entre  $-2^{31}$  e  $2^{31} - 1$ )
- O tipo **SMALLINT** utiliza 2 bytes (números entre  $-2^{15}$  e  $2^{15} - 1$ )
- O tipo **TINY** utiliza 1 bytes (números 0 e 255)
- O tipo **BIT** armazena 0,1 ou NULL



---

## ***Caso prático - Alguns Tipos de dados no SQL Server (cont.)***

---

- O tipo **FLOAT** pode armazenar valores compreendidos entre  $-1.79E + 308$  e  $1.79E + 308$ , quando é especificado um  $n$  de 53 (máximo)
- Quando  $1 \leq n \leq 24$ , tem-se uma precisão de 7 dígitos e ocupa 4 bytes
- Quando  $25 \leq n \leq 53$ , tem-se uma precisão de 15 dígitos e ocupa 8 bytes
  
- **FLOAT(24)** é sinónimo **REAL**
- **FLOAT(53)** é sinónimo de **DOUBLE PRECISION**

**É de evitar a referência a colunas “floating-point”  
em cláusulas WHERE**



---

## ***Caso prático - Alguns Tipos de dados no SQL Server (cont.)***

---

- O tipo XML dá suporte a dados do tipo XML. Não é comparável, por isso tem algumas limitações no seu uso
  - Nomeadamente: não pode ser PRIMARY KEY ou FOREIGN KEY
- O tipo TIMESTAMP, ao contrário do seu nome, não é uma estampilha temporal. O seu uso está DEPRECATED e a alternativa é a utilização do tipo ROWVERSION.
  - O valor que este tipo tem é binário, gerado de forma automática pelo sistema.
  - Apenas pode existir uma coluna deste tipo por tabela
- O tipo **UNIQUEIDENTIFIER** serve para armazenar GUIDs (Global Unique Identifiers)
  - Tem o formato xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx , onde cada x é um valor hexadecimal.

---

## Conversão de tipos

---

- Por vezes é necessário efectuar a uma conversão de tipos explícita
  - Um determinado atributo é do tipo *datetime*, que pode não ter equivalente numa determinada linguagem usada para manipular os resultados da interrogação
  - Neste caso é necessário efectuar uma conversão de tipos explícita
- Em SQL utiliza-se a função CAST para efectuar essa conversão
- A sintaxe do CAST:
  - CAST(<expressão> AS <tipo de dados>)
- Questão: qual o código e o nome dos empregados e a sua data de admissão (com as datas como *char(20)*)

```
SELECT codEmp, nome, CAST(dataAdmissao as CHAR(20))  
FROM EMPREGADO
```

---

## ***Mapeamento de valores***

---

- Existem casos em que é necessário mapear os valores existentes num determinado domínio para um outro:
  - Um determinado atributo é do tipo real mas para um determinado caso é necessário passar esse valor para um conjunto de valores discretos
  - Pode utilizar-se o CASE para efectuar esse mapeamento
- A Sintaxe do CASE:
  - CASE WHEN <condição> THEN <valor> ELSE <valor>
- Questão: Qual a classificação para cada disciplina do aluno 123?

```
SELECT disciplina,  
CASE  
    WHEN E.nota <10 THEN 'Reprovado'  
    ELSE 'Aprovado'  
END as nota  
FROM EMPREGADO as E
```

---

## ***LMD – comandos de manipulação de dados***

---

- A LMD, não só permite aceder à informação (SELECT), como também permite alterá-la e actualizá-la
- Existem mais três comandos que permitem manipular a informação:

- INSERT (insere novas linhas numa tabela)

```
INSERT INTO <nome da tabela> [(coluna1, coluna2, ...)]  
VALUES (valor1, valor2, ...) | <comando SELECT>
```

- UPDATE (actualiza linhas de uma tabela)

```
UPDATE <nome da tabela>  
SET coluna = valor | expressão, coluna = valor |  
expressão, ...  
[WHERE <condição>]
```

- DELETE (remove linhas de uma tabela)

```
DELETE FROM <nome da tabela> [WHERE <condição>]
```

---

# ***INSERT***

---

- O comando INSERT é usado para inserir dados numa determinada tabela
- **Exemplo:** Pretende-se inserir informação na tabela CLIENTE(BI,nome), introduzindo a informação do cliente José Maria com BI 123

```
INSERT INTO CLIENTE(NOME,BI) VALUES('José Maria', 123)
```

- Nos casos em que o comando INSERT é usado desta forma, apenas é inserido uma linha de cada vez na tabela
  - Quando é omitida a lista dos nomes das colunas, os valores tem de ser dados segundo a ordem das colunas definidas na tabela
  - Quando é especificada a lista de nomes, os valores tem de estar de acordo com essa lista, embora esta não esteja, necessariamente, pela ordem definida na tabela
  - Quando são omitidos valores, é assumido que têm o valor NULL

---

## ***INSERT (cont.)***

---

- Pretende-se copiar para a tabela com informação histórica dos empregados, todos os empregados dos departamentos de Lisboa. O Esquema de Relação que irá conter a informação histórica dos empregados é:
  - HISTORICO\_EMPREGADO( codEmp, nome )

```
INSERT INTO HISTORICO_EMPREGADO( codEmp, nome )  
  SELECT codEmp, E.nome  
  FROM EMPREGADO AS E  
    INNER JOIN DEPARTAMENTO AS D  
  ON (E.codDep = D.codDep )  
  WHERE localizacao = 'Lisboa'
```

- Nos casos em que o comando INSERT é usado desta forma
  - o número de colunas na lista da cláusula SELECT tem que ser igual ao número de colunas referidas no comando INSERT e os domínios de colunas correspondentes têm que ser compatíveis

---

## **UPDATE**

---

- O comando **UPDATE** é usado para alterar os dados nas linhas de uma determinada tabela
- Pretende-se registar o facto do empregado com código 123 ter mudado para o departamento 444 cujo chefe tem o código 654

```
UPDATE EMPREGADO  
SET codDep = 444, codEmpChefe = 654  
WHERE codEmp = 123
```

- Características do comando UPDATE:
  - se a cláusula WHERE for omitida, todas as linhas da tabela são actualizadas
  - os valores a actualizar podem ser o resultado de expressões ou interrogações à Base de Dados



---

## ***DELETE***

---

- O comando DELETE é usado para remover linhas de uma determinada tabela
- Pretende-se remover os empregados do departamento com o código 444

```
DELETE FROM EMPREGADO  
WHERE codDep=444
```

- Pretende-se remover toda a informação relativa ao histórico dos empregados

```
DELETE FROM HISTORICO_EMPREGADO
```

- Características do comando DELETE:
  - se a cláusula WHERE for omitida, todas as linhas da tabela são removidas

---

## ***LDD – Linguagem de definição de dados***

---

- Existem três comandos pertencentes á LDD:
  - CREATE (criar estruturas de dados)
  - ALTER (alterar estruturas de dados)
  - DROP (remover estruturas de dados)
- A criação de tabelas pode ser efectuada em qualquer momento de uma sessão SQL
- A estrutura de uma tabela pode ser alterada em qualquer momento de uma sessão SQL, podendo ser perdida a informação anteriormente armazenada nessa tabela
- Uma tabela não tem qualquer dimensão pré determinada

---

## ***CREATE – para criar uma tabela***

---

- A sintaxe do geral do CREATE, para este caso:

```
CREATE TABLE <nome tabela>
( {<nome coluna> <tipo>
    [DEFAULT <valor | função | NULL>]
    [<restrição de coluna>]
  }+
  [restrição de tabela]
);
```

- Onde
  - <restrição de coluna> indica uma restrição a aplicar a uma coluna
  - <restrição de tabela> aplica-se a mais de uma coluna
- Cada definição é separada por virgula

---

## ***CREATE – para criar uma tabela (cont.)***

---

- As restrições de tabela podem ser

`[CONSTRAINT nome_restrição]`

`[{PRIMARY KEY | UNIQUE}] ( coluna, ... ) |`

`[FOREIGN KEY ( coluna, ... ) {REFERENCES tabela [( coluna, ... )]`

`[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}] |`

`[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]]`

`[CHECK ( condição )]`

- As restrições definidas na criação das tabelas são asseguradas pelo SGDB durante a manipulação dos dados (INSERT, UPDATE, DELETE)
- Quando alguma restrição for violada, o comando em execução é abortado

---

## ***CREATE – para criar uma tabela (cont.)***

---

- Definição de uma tabela com uma chave primária, uma chave candidata (alternativa) e restrição de valor NULL:

```
CREATE TABLE ALUNO(  
    numAluno int CONSTRAINT pk_ALUNO PRIMARY KEY,  
    numBI char( 8 ) NOT NULL CONSTRAINT ak1_ALUNO UNIQUE,  
    nome varchar( 100 ),  
    CONSTRAINT ck1_ALUNO CHECK ( nome IS NOT NULL )  
)
```

- Definição de uma tabela com chave primária composta e restrição de valor NULL:

```
CREATE TABLE FACTURA(  
    numFactura int,  
    numLinha int,  
    quantidade int NOT NULL,  
    CONSTRAINT pk_FACTURA PRIMARY KEY ( numFactura, numLinha )  
)
```

---

## ***CREATE – para criar uma tabela (cont.)***

---

- Definição de uma tabela com chave estrangeira composta:

```
CREATE TABLE ENTREGA
(
    numEntrega int,
    dataEntrega datetime NOT NULL,
    numEnc int NOT NULL,
    numProduto int NOT NULL,
    CONSTRAINT pk_ENTREGA PRIMARY KEY ( numEntrega ),
    CONSTRAINT fk1_ENTREGA FOREIGN KEY (numEnc , numProduto )
    REFERENCES FACTURA ( numFactura, numLinha )
    ON DELETE CASCADE
)
```

- ON DELETE CASCADE** indica que a remoção de uma linha da tabela FACTURA implica a remoção das linhas da tabela ENTREGA que lhe estiverem associadas

---

## ***CREATE – para criar uma tabela (cont.)***

---

- Definição de uma tabela com uma regra de verificação:

```
CREATE TABLE ENCOMENDA
(  numEnc int CONSTRAINT pk_ENCOMENDA PRIMARY KEY,
   dataEnc datetime NOT NULL,
   codCliente int NOT NULL CONSTRAINT fk1_ENCOMENDA
  FOREIGN KEY REFERENCES CLIENTE ( codCliente ),
   dataEntrega datetime,
   CONSTRAINT ck1_ENCOMENDA CHECK ( dataEntrega >
dataEnc )
)
```

---

## ***ALTER – para alterar uma tabela***

---

- Sintaxe do comando ALTER, para alteração de tabelas:

```
ALTER TABLE nome_tabela  
    [ ADD {[COLUMN] novas colunas | CONSTRAINT <novas  
    restrições_coluna>}]  
    [ ALTER [COLUMN] coluna]  
    [ DROP {[COLUMN] coluna} | {CONSTRAINT restrição_coluna} [RESTRICT  
    | CASCADE]]
```

- As alterações possíveis a uma tabela são:
  - acrescentar colunas, eventualmente acompanhadas de restrições
  - alterar a definição de colunas existentes
  - acrescentar restrições de integridade à tabela
  - remover uma restrição da tabela
- Não é possível, no entanto:
  - modificar uma coluna com valores NULL para NOT NULL.
  - Só se pode adicionar uma coluna NOT NULL a uma tabela que não contenha nenhuma linha.
  - Remover uma coluna se essa for a única existente na tabela



---

## ***ALTER – para alterar uma tabela (cont.)***

---

- Alguns exemplos

- Adição de uma coluna:

```
ALTER TABLE EMPREGADO ADD comissao int NOT NULL
```

- Modificação da definição de uma coluna:

```
ALTER TABLE EMPREGADO ALTER column comissao smallint  
NOT NULL
```

- Remoção de uma coluna:

```
ALTER TABLE EMPREGADO DROP comissao
```

- Eliminar uma restrição de integridade:

```
ALTER TABLE ENCOMENDA DROP CONSTRAINT ck1_ENCOMENDA
```

- Acrescentar uma restrição de integridade (chave estrangeira):

```
ALTER TABLE ENCOMENDA ADD CONSTRAINT fk1_ENCOMENDA FOREIGN  
KEY codCliente REFERENCES CLIENTE ( codCliente )
```

---

## ***DROP – para remover uma tabela***

---

- A sintaxe do DROP, para remover tabelas
  - DROP TABLE <nome-tabela> [RESTRICT|CASCADE]
- Exemplo de remoção da tabela EMPREGADO

```
DROP TABLE EMPREGADO  
DROP TABLE EMPREGADO RESTRICT
```
- Algumas características da acção de remoção de uma tabela:
  - a remoção de uma tabela causa a perda de todos os dados nela existentes, assim como de todos os índices associados
  - uma tabela só pode ser removida por quem a criou ou pelo administrador da Base de Dados
  - se a tabela estiver a ser referenciada em VIEWS (abordadas mais adiante) ou em restrições de integridade, o comando DROP falha
  - no entanto se for especificada a palavra CASCADE, tanto a tabela como quem a referencia (VIEWS e restrições de integridade) são removidas

---

## ***VIEWS – Vistas sobre os dados***

---

- Por vezes, é necessário, por razões de segurança ou de simplicidade, criar “tabelas virtuais” que apresentam os dados numa forma diferente daquela segundo a qual estes estão armazenados
- Usando a terminologia SQL, uma Vista (View) consiste numa única estrutura de dados construída a partir de uma interrogação a:
  - tabelas
  - Vistas anteriormente definidas
- Uma Vista apesar de poder ser manipulada como uma tabela, não tem armazenamento próprio, o que:
  - origina algumas limitações às operações de actualização (update)
  - mas não limita as operações de interrogação (select)

---

## ***VIEWS (cont.)***

---

- Podem ser consideradas dois tipos de Vistas, de acordo com a interrogação que as define
  - Vistas simples:
    - construídas com base numa única tabela, não contêm funções nem grupos de dados
  - Vistas complexas:
    - construídas com base em várias tabelas, contêm funções ou grupos de dados
- Utilidade das Vistas:
  - mostrar apenas parte dos dados ( segurança)
  - permitir que os mesmos dados sejam visualizados de diferentes maneiras por diferentes utilizadores ( segurança)
  - simplificar a consulta dos dados, substituindo consultas elaboradas envolvendo várias tabelas, por fáceis consultas sobre a Vista
  - reduzir a possibilidade de incoerências (WITH CHECK OPTION)

---

## ***VIEWS (cont.)***

---

- Sintaxe do CREATE, para criação de vistas:

```
CREATE VIEW <nome_vista>  
    [ (nome_coluna_1, nome_coluna_2, ...) ]  
    AS comando_select [WITH CHECK OPTION]
```

- nome\_coluna\_i
  - nome da coluna usado na vista. Se não for especificado é assumido o mesmo nome das colunas definidas na cláusula SELECT
- A directiva SELECT tem algumas limitações
  - não pode incluir as cláusulas ORDER BY
  - não pode incluir a instrução INTO (*select ... into ... tabela*)
  - não pode referenciar uma tabela temporária
  - não pode referenciar a própria vista (recursividade!)

---

## ***VIEWS (cont.)***

---

- Seja:
  - DEPARTAMENTO ( nome, localizacao )
  - EMPREGADO ( cod, nome, salario, nomeDep )
- Pretende-se criar uma vista que permita saber:
  - Para cada departamento quantos empregados existem e qual o montante total de salários

```
CREATE VIEW INFORMACAO_DEPARTAMENTO
(nomeDepartamento, numEmpregados , totalSalarios )
AS SELECT D.nome, COUNT(*), SUM(salario)
FROM DEPARTAMENTO AS D INNER JOIN EMPREGADO AS E
ON ( D.nome = E.nomeDep )
GROUP BY D.nome
```

---

## ***VIEWS (cont.)***

---

- Pretende-se, utilizando a vista INFORMACAO\_DEPARTAMENTO, para responder à questão: Para o departamento de Informática, quantos empregados existem e qual o montante total de salários

```
SELECT numEmpregados , totalSalarios  
FROM INFORMACAO_DEPARTAMENTO  
WHERE nomeDepartamento = 'Informática'
```

- Algumas considerações:
  - A vista não é concretizada no momento em que é criada, mas sempre que é especificada uma interrogação sobre essa vista, mantendo-se assim sempre actualizada
  - As alterações das tabelas originais reflectem-se nas diversas vistas onde essas tabelas são referenciadas

---

## *Remoção de vistas*

---

- A remoção de vistas é feita utilizando o comando DROP, com a sintaxe:
  - DROP VIEW <nome vista> [RESTRICT | CASCADE]
- Exemplo de remoção da vista INFORMACAO\_DEPARTAMENTO  
`DROP VIEW INFORMACAO_DEPARTAMENTO`
- Algumas considerações:
  - a remoção de uma vista não tem qualquer influência nos dados das tabelas que lhe serviam de base
  - se existirem outras vistas que dependam da vista removida
    - A acção falha (dependendo da implementação de cada SGBD)
    - É especificado CASCADE e essas vistas são igualmente removidas
  - uma vista só pode ser removida por um utilizador com permissões para efectuar essa operação (ou pelo SA)



---

## ***Actualização de dados sobre vistas***

---

- Se a vista for definida sobre uma única tabela e sem funções de agregação de dados:
  - é uma operação simples que se traduz na actualização da tabela que lhe serve de base
- No entanto se a vista envolver múltiplas tabelas e funções de agregação de dados:
  - é uma operação complicada e que pode ser ambígua
- De uma forma geral, não são actualizáveis as vistas:
  - definidas sobre múltiplas tabelas utilizando junções (join)
  - que utilizam agrupamento de dados e funções de agregação
  - Que utilizam operações sobre álgebricas, por exemplo UNION

---

## ***Actualização de dados sobre vistas (cont.)***

---

- Alguns exemplos:
  - O comando de DELETE não é permitido se a vista incluir:
    - condições de junção (join)
    - funções de agrupamento
    - o comando DISTINCT
    - sub-interrogações correlacionadas
  - O comando de UPDATE não é permitido se a vista incluir:
    - qualquer das limitações comando de DELETE
    - colunas definidas por expressões (ex: salarioAno = 14 \* salario)
  - O comando de INSERT não é permitido se a vista incluir:
    - qualquer das limitações do comando UPDATE;
    - colunas com possibilidade de terem valores NOT NULL que não tenham valores de omissão nas tabelas base e que não estejam incluídas na vista através da qual se pretende inserir novas colunas

---

## ***Actualização de dados sobre vistas (cont.)***

---

- Os comandos de INSERT e UPDATE são permitidos em vistas contendo várias tabelas base se:
  - o comando afectar apenas uma das tabelas que serve de base à vista
- Se as vistas forem criadas com a opção WITH CHECK OPTION, algumas das alterações podem não ser possíveis
  - Esta opção só permite INSERTs ou UPDATEs sobre vistas se, finalizadas essas acções, o resultado seja visível na vista
  - Por outras palavras, as alterações têm de ser compatíveis com as condições especificadas na cláusula WHERE
  - Se tal não acontecer, as alterações não são permitidas e o comando é abortado

---

## ***Actualização de dados sobre vistas (cont.)***

---

- Pretende-se criar uma vista que permita saber:
  - Quais os empregados que têm um salário inferior a 250000.
  - Também se pretende que qualquer acção de alteração sobre essa vista apenas afecte os empregados cujo salário seja inferior a 250000

- Ou seja

```
CREATE VIEW VISTA_EMPREGADO  
AS SELECT cod, nome FROM EMPREGADO  
WHERE salario < 250000  
WITH CHECK OPTION
```

- As instruções de INSERT e UPDATE sobre esta vista têm que verificar sempre a condição definida na cláusula WHERE

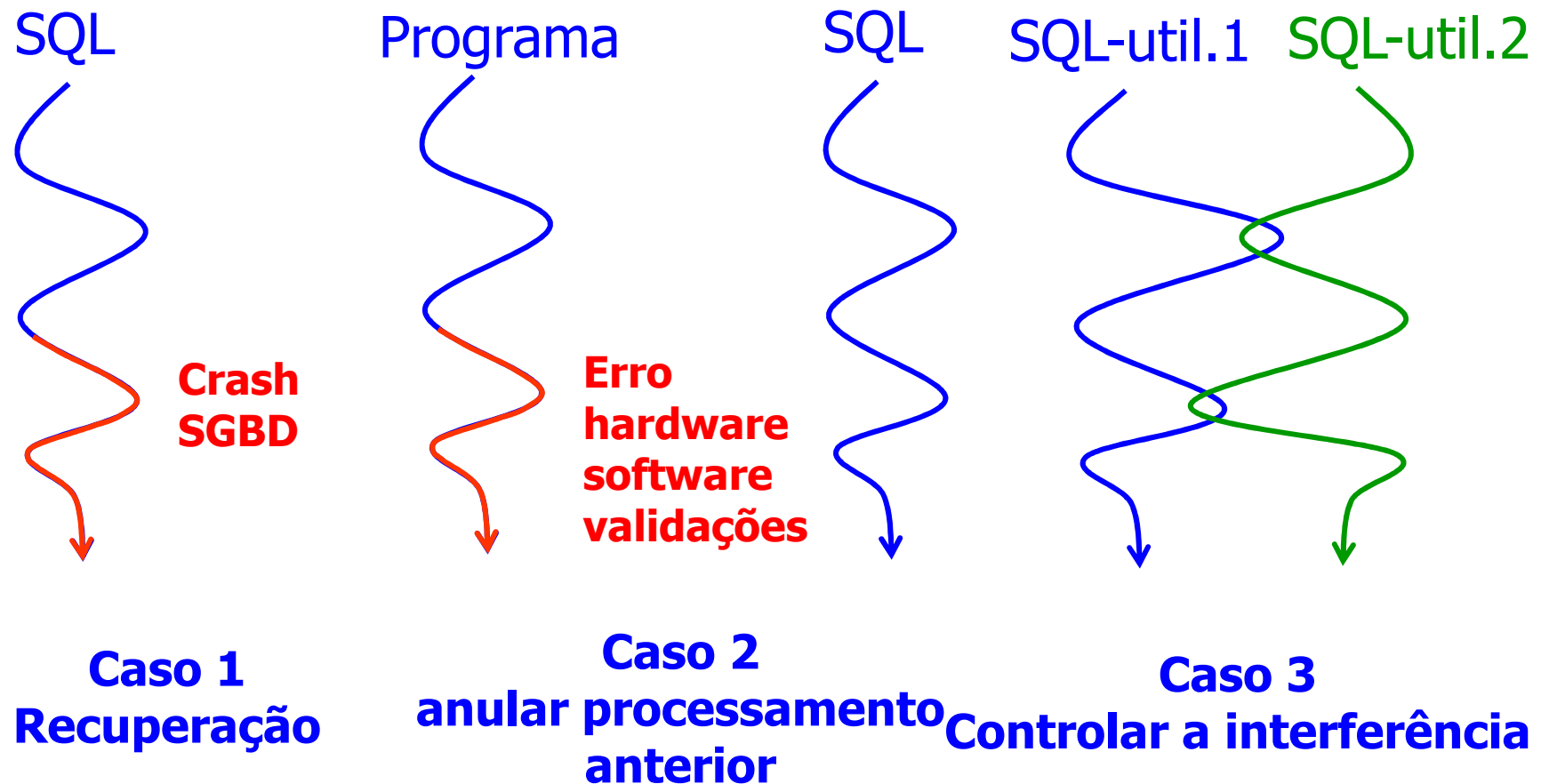
---

## ***Tratamento da vistas no SGDB***

---

- O comando CREATE VIEW não origina a execução do comando SELECT a ele associado
- O comando CREATE VIEW apenas origina o armazenamento da definição da vista (directiva SELECT) no dicionário de dados
- Ao aceder aos dados através de uma vista, o sistema:
  - Extrai a definição da vista do dicionário de dados
  - verifica as permissões de acesso à vista
  - converte a operação sobre a vista numa operação equivalente na tabela ou tabelas que servem de base à vista (grau o plano de execução)

## Processamento transaccional - Motivação



---

## Propriedades das Transacções

---

- As transacções tem um conjunto de propriedades, normalmente designadas de **ACID**
- **Atomicidade (*Atomicity*)**
  - Uma transacção é indivisível no seu processamento
  - Ou todas as instruções dentro de uma transacção são executados ou nenhuma o é
- **Consistência (*Consistency Preservation*)**
  - A execução de uma transacção leva a base de dados de um estado consistente para outro estado consistente
- **Isolamento (*Isolation*)**
  - As acções efectuadas por uma transacção só devem ser visíveis para outras transacções depois desta ter sido concluída com êxito
- **Durabilidade (*Durability ou Permanency*)**
  - Depois de uma transacção ter sido efectuada com sucesso o seu resultado é persistente, mesmo se existirem eventuais falhas posteriormente

**No âmbito da UC de Sistemas de Informação 1  
apenas será tratada a Atomicidade**



---

## Exemplo

---

- Considere-se os seguintes troços de código

```
INSERT INTO CLIENTE(NOME,BI) VALUES('José Maria', 123),  
( 'Maria José', 321);
```

```
INSERT INTO CLIENTE(NOME,BI) VALUES('José Maria', 123);  
INSERT INTO CLIENTE(NOME,BI) VALUES('Maria José', 321);
```

- No primeiro caso, se existir um erro na inserção de um tuplo, não será introduzida nenhuma informação na tabela cliente
- No segundo caso, se existir um erro no inserção, por exemplo, do cliente Maria José, a informação do primeiro cliente ficará na tabela cliente.
- Isto deve-se ao facto de que uma única instrução SQL é efectuada dentro de uma transacção, garantindo-se uma execução atómica

**Como tornar os dois troços SQL são equivalentes?**



---

## *Controlo transaccional*

---

- Uma só acção (instrução SQL) é sempre vista como uma transacção
- Para garantir que um conjunto de instruções passa a ser atómico é normalmente necessário iniciar a transacção explicitamente\*
  - Através do código

`BEGIN TRANSACTION nomeTransacção`

- Uma vez efectuado o processamento transaccional, é necessário indicar como deve ser terminado:
  - Consolidando (validando) as acções efectuadas, com  
`COMMIT TRANSACTION nomeTransacção`
  - Ou desfazendo as acções efectuadas, com  
`ROLLBACK TRANSACTION nomeTransacção`

**\*Existem formas de iniciar implicitamente transacções.**

**Também é possível guardar resultados parciais (Savepoints) de processamentos transaccionais.**



---

## *Exemplo revisitado*

---

- Então para garantir que as duas inserções são tratadas como um bloco indivisível é necessário iniciar uma transacção

```
BEGIN TRANSACTION
INSERT INTO CLIENTE(NOME,BI) VALUES('José Maria', 123);
INSERT INTO CLIENTE(NOME,BI) VALUES('Maria José', 321);
COMMIT TRANSACTION
```

- No entanto, a construção anterior pode não ter o resultado esperado.
- Se existir um erro na segunda inserção, esta é abortada, mas o processamento continua, executando-se o COMMIT e validando a transacção
- É necessário garantir que a transacção é abortada quando existe um erro
  - Uma solução é recorrer ao controlo estruturado de erros

---

## *Tratamento estruturado de erros – SQL Server*

---

- No standard é possível definir handlers de tratamento de erros
- No caso prático do SQL Server existe a construção

```
BEGIN TRY
    <instruções>
END TRY
BEGIN CATCH
    <instruções de tratamento do erro>
END CATCH
```

- Note-se que apenas são apenas os erros com gravidade ]10, 20 [ são “apanhados” no bloco CATCH
- Erros com gravidade inferior são avisos
- Erros com gravidade superior são fatais e termina a ligação, terminando igualmente as transacções que estiverem a decorrer no âmbito desta.

---

## *Tratamento estruturado de erros – SQL Server (cont.)*

---

- Assim, o exemplo anterior pode ser reescrito da seguinte forma:

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO CLIENTE(NOME,BI) VALUES('José Maria', 123);
        INSERT INTO CLIENTE(NOME,BI) VALUES('Maria José', 321);
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
END CATCH
```

- No entanto, o utilizador não será avisado do erro existente no processamento, uma vez que o erro foi tratado
- A solução passa por “deixar” o erro continuar até ao utilizador

---

## *Tratamento estruturado de erros – SQL Server (cont.)*

---

- Uma possibilidade é utilizar o comando RAISEERROR para reportar o erro

```
RAISERROR (msg_str, severity, state )
```

- Ou seja

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO CLIENTE(NOME,BI) VALUES('José Maria', 123);
        INSERT INTO CLIENTE(NOME,BI) VALUES('Maria José', 321);
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
    RAISEERROR(ERROR_MESSAGE(), ERROR_SEVERITY(), ERROR_STATE( ))
END CATCH
```

- Onde ERROR\_MESSAGE (), ERROR\_SEVERITY (), ERROR\_STATE () retornam a informação relativa ao erro que fez correr o bloco catch

---

## *Exemplo re-revisitado*

---

- No SQL Server existe uma solução mais simples para abortar uma transacção quando acontece um erro
  - Basta activar a opção XACT\_ABORT

```
SET XACT_ABORT ON
```

```
BEGIN TRANSACTION
```

```
    INSERT INTO CLIENTE(NOME,BI) VALUES('José Maria', 123);
```

```
    INSERT INTO CLIENTE(NOME,BI) VALUES('Maria José', 321);
```

```
COMMIT TRANSACTION
```

- Os erros de compilação não afectam a propriedade, i.e., não causam o termino da transacção com ROLLBACK