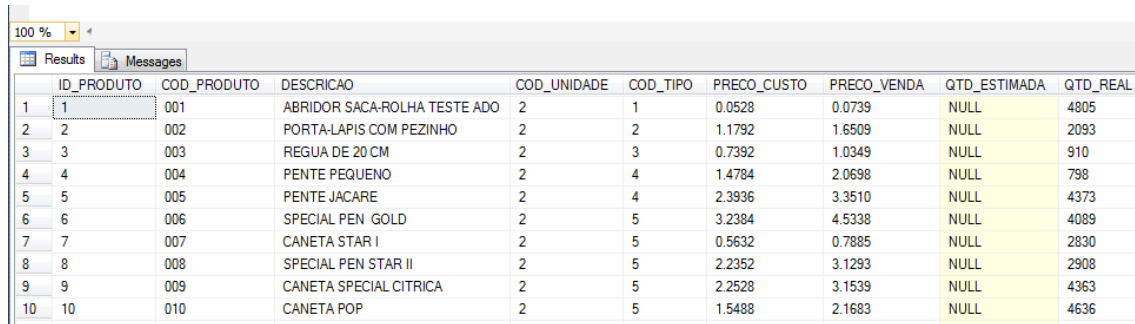




- **Banco de dados relacional**

Um banco de dados é a forma organizada de armazenar informações de modo a facilitar sua inserção, alteração, exclusão e recuperação.

No banco relacional as tabelas são armazenadas em tabelas retangulares, semelhante a uma planilha. Na maioria das vezes as tabelas possuem uma informação chave que as relaciona.



	ID_PRODUTO	COD_PRODUTO	DESCRICAO	COD_UNIDADE	COD_TIPO	PRECO_CUSTO	PRECO_VENDA	QTD_ESTIMADA	QTD_REAL
1	1	001	ABRIDOR SACA-ROLHA TESTE ADO	2	1	0.0528	0.0739	NULL	4805
2	2	002	PORTA-LAPIS COM PEZINHO	2	2	1.1792	1.6509	NULL	2093
3	3	003	REGUA DE 20 CM	2	3	0.7392	1.0349	NULL	910
4	4	004	PENTE PEQUENO	2	4	1.4784	2.0698	NULL	798
5	5	005	PENTE JACARE	2	4	2.3936	3.3510	NULL	4373
6	6	006	SPECIAL PEN GOLD	2	5	3.2384	4.5338	NULL	4089
7	7	007	CANETA STAR I	2	5	0.5632	0.7885	NULL	2830
8	8	008	SPECIAL PEN STAR II	2	5	2.2352	3.1293	NULL	2908
9	9	009	CANETA SPECIAL CITRICA	2	5	2.2528	3.1539	NULL	4363
10	10	010	CANETA POP	2	5	1.5488	2.1683	NULL	4636

- **Linguagens SQL e T-SQL**

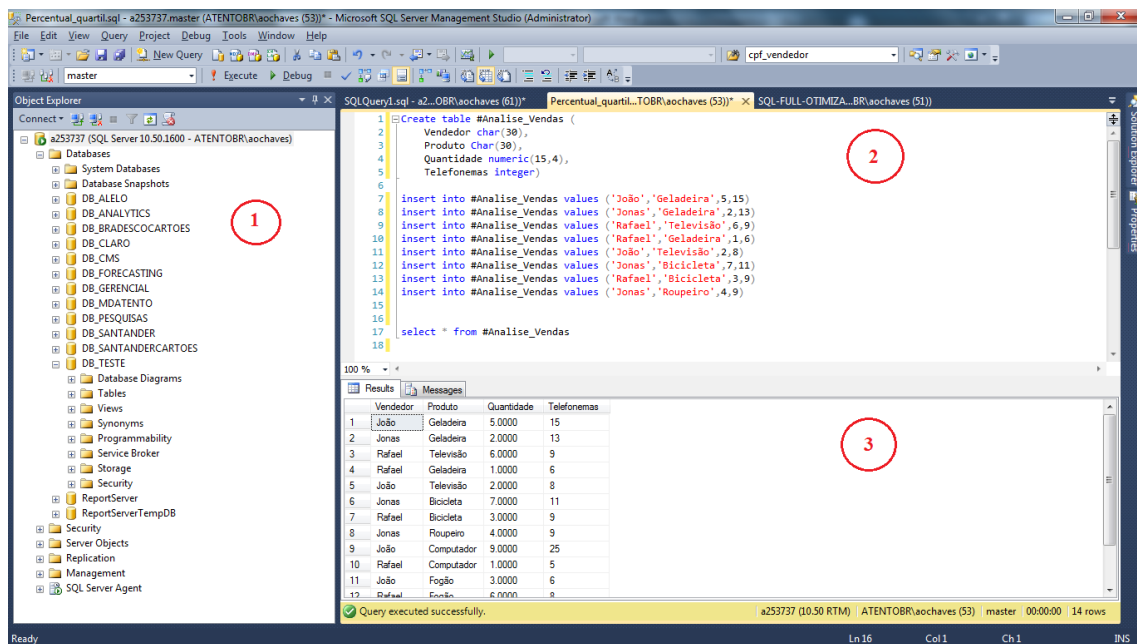
Toda a manipulação de um banco de dados é feita através de uma linguagem específica, com exigência sintática rígida, chamada SQL (Structure Query Language). Os fundamentos desta linguagem estão baseados no conceito de banco de dados relacional.

Esta linguagem foi desenvolvida pela IBM no início da década de 70, e posteriormente adotada como linguagem padrão pela ANSI (American National Standard Institute) e pela ISSO (International Organization for Standardization), em 1986 e 1987, respectivamente.

A T-SQL (Transact-SQL) é uma implementação da Microsoft para a SQL padrão ANSI. Ela cria opções adicionais para os comandos e também cria novos comandos que permitem o recurso de programação, como os controles de fluxo, variáveis de memória e etc.

- **SQL Server Management Studio**

O SQL Server Management Studio é um aplicativo de software usado para configurar, gerenciar e administrar todos os componentes dentro Microsoft SQL Server. A ferramenta inclui ambos os editores de script e ferramentas gráficas que trabalham com objetos e recursos do servidor.



1 - Object Explorer - permite ao usuário navegar, selecionar, e agir de acordo com qualquer um dos objetos dentro do servidor.

2 – Code Editor – Permite escrever comandos T-SQL. Cada conexão possui um número de sessão único.

3 – Área que exibe o resultado da consulta que fora executada.

• Tabela

Uma tabela pode ser entendida como um conjunto de linhas e colunas. As colunas de uma tabela qualificam cada elemento (linha) com informações relacionadas ao objeto.

Utilizando esses conceitos, é possível armazenar dados em uma ou várias tabelas, dependendo do que e como desejamos as informações.

• Numéricos exatos

Bit - Um tipo de dados inteiro que pode aceitar um valor 1, 0 ou NULL.

TINYINT - Valores numéricos inteiros variando de 0 até 256

INT - Valores numéricos inteiros variando de -2.147.483.648 até 2.147.483.647

BIGINT - O tipo de dados bigint deve ser usado quando valores inteiros possam exceder o intervalo ao qual tipo de dados int oferece suporte. Valores numéricos inteiros variando de -92.23.372.036.854.775.808 até 9.223.372.036.854.775.807

SMALLINT - Valores numéricos inteiros variando de -32.768 até 32.767

DECIMAL (I, D) e NUMERIC (I, D) - Armazenam valores numéricos inteiros com casas decimais utilizando precisão. I deve ser substituído pela quantidade de dígitos total do número e D deve ser substituído pela quantidade de dígitos da parte decimal (após a vírgula). DECIMAL e NUMERIC possuem a mesma funcionalidade, porém DECIMAL faz parte do padrão ANSI e NUMERIC é mantido por compatibilidade. Por exemplo, DECIMAL(8,2) armazena valores numéricos decimais variando de - 999999,99 até 999999,99

SMALLMONEY - Valores numéricos decimais variando de -214.748,3648 até 214.748,3647

MONEY - Valores numéricos decimais variando de -922.337.203.685.477,5808 até 922.337.203.685.477,5807

Os tipos de dados money e smallmoney são precisos em dez milésimos de unidades monetárias que representam.

Use um ponto para separar unidades monetárias parciais, como centavos, de unidades monetárias inteiras. Por exemplo, 2.15 especifica 2 dólares e 15 centavos.

- **Numéricos aproximados**

REAL – Valores numéricos aproximados com precisão de ponto flutuante, indo de -3.40E+ 38 até 3.40E + 38

FLOAT - Valores numéricos aproximados com precisão de ponto flutuante, indo de -1.79E + 308 até 1.79E + 308

Os dados de ponto flutuante são aproximados; portanto, nem todos os valores no intervalo de tipo de dados podem ser representados de maneira exata.

- **Data e hora**

TIME – Define uma hora de um dia. A hora se encontra sem reconhecimento de fuso horário e se baseia em um relógio de 24 horas. Varia de 00:00:00 até 23:59:59.9999999.

DATE – Armazena data variando de 01/01/0001 até 31/12/9999. Com precisão de 100 nano segundos.

SMALLDATETIME - Armazena hora e data variando de 1 de janeiro de 1900 até 6 de junho de 2079. Com precisão de 1 minuto.

DATETIME - Armazena hora e data variando de 1 de janeiro de 1753 até 31 de Dezembro de 9999. Com precisão de 3.33 milissegundos.

DATETIME2 – Armazena hora e data variando de 01/01/0001 00:00:00 até 31/12/9999 23:59:59.9999999.

DATETIMEOFFSET – Armazena data variando de 01/01/0001 até 31/12/9999. Com precisão de 100 nano segundos e com indicação de fuso horário, cujo intervalo pode variar de -14:00 a +14:00.

- **Cadeias de caracteres**

CHAR(N) - Armazena N caracteres fixos (até 8.000) no formato não Unicode. Se a quantidade de caracteres armazenada no campo for menor que o tamanho total especificado em N, o resto do campo é preenchido com espaços em branco.

VARCHAR (N | MAX) - Armazena N caracteres (até 8.000) no formato não Unicode. Se a quantidade de caracteres armazenada no campo for menor que o tamanho total especificado em N, o resto do campo não é preenchido. MAX indica que o tamanho de armazenamento máximo é $2^{31}-1$ bytes (2 GB)

TEXT - Armazena caracteres (até 2.147.483.647) no formato não Unicode. Se a quantidade de caracteres armazenada no campo for menor que 2.147.483.647, o resto do campo não é preenchido. Procure não utilizar este tipo de dado diretamente, pois existem funções específicas para trabalhar com este tipo de dado.

- **Comandos SQL**

A linguagem SQL é constituída de diferentes tipos de comandos e estes comandos são agrupados em três categorias principais, sendo comandos:

- **Comandos DDL - Data Definition Language** as instruções que definem estruturas de armazenamento dos dados.

CREATE DATABASE

Database é o conjunto de arquivos que armazena todos os objetos do banco de dados. No máximo 32.767 bancos de dados podem ser especificados em uma instância do SQL Server.

CREATE DATABASE Nome banco de dados

CREATE TABLE

O comando CREATE TABLE cria uma tabela, inicialmente vazia, no banco de dados corrente. O usuário que executa o comando se torna o dono da tabela.

```
CREATE TABLE Nome tabela (  
  
    -- cria um campo de cadeira numérica int e seta coluna como primary key  
    RE int primary key NOT NULL,  
  
    -- cria um campo de texto de no máximo 30 caracteres  
    NOME CHAR(30),  
  
    -- cria um campo com numero inteiro com restrição de registros vazios  
    [LOGIN] int NOT NULL,  
  
    REFERENCES CMS.[LOGIN](ProductID), --cria chave estrangeira com referência a tabela x  
  
    -- cria campo do tipo money  
    SALARIO money NULL,  
  
    -- cria campo do tipo data  
    DATANASC datetime NULL,  
  
    -- cria um campo tipo data e hora  
    DATAINCLUSAO datetime NOT NULL --  
  
    DEFAULT (getdate()), -- traz como padrão a data de inserção do dado  
  
    -- cria coluna calculada  
    IDADE AS ((DATAINCLUSAO-GETDATE())));
```

ALTER TABLE

Modifica uma definição de tabela alterando, adicionando ou removendo colunas e restrições.

-- O exemplo a seguir adiciona uma nova coluna com uma restrição UNIQUE.

```
ALTER TABLE Nome tabela ADD ENDERECO VARCHAR(20) NULL UNIQUE;
```

DROP TABLE

Remove uma ou mais definições de tabela e todos os dados, índices, gatilhos, restrições e especificações de permissão dessas tabelas.

DROP TABLE Nome tabela

TRUNCATE

Remove todas as linhas de uma tabela sem registrar as exclusões de linhas individuais. TRUNCATE TABLE é semelhante à instrução DELETE sem nenhuma cláusula WHERE; entretanto, TRUNCATE TABLE é mais rápida e utiliza menos recursos de sistema e log de transações.

TRUNCATE TABLE Nome tabela

- **Comandos DML - Data Manipulation Language** os comandos que manipulam os dados

SELECT

O Select é o principal comando usado em SQL para realizar consultas a dados pertencentes a uma tabela.

*SELECT * FROM Nome tabela*

INSERT INTO

O comando INSERT é classificado como posicional quando não especifica a lista de colunas que receberão os dados de VALUES. Neste caso a lista de valores precisa ter todos os campos, exceto o IDENTITY, na ordem física em que foram criadas no comando CREATE TABLE.

INSERT INTO Nome tabela

VALUES('440542','30544','1000','2015-01-01','50')

O INSERT é classificado como declarativo quando especifica as colunas que receberão os dados da lista de valores.

INSERT INTO Nome tabela (RE, [LOGIN], SALARIO, DATANASC, IDADE)

VALUES ('440542','30544','1000','2015-01-01','50')

UPDATE

Uma vez que uma linha esteja na tabela, pode-se querer alterar o conteúdo de uma ou mais colunas, ou até o conteúdo de uma coluna em diversas linhas. É de suma importância a utilização da cláusula WHERE caso necessário.

UPDATE Nome tabela

SET RG = '1020304050'

WHERE RE = '440545'

DELETE

Remove uma ou mais linhas de uma tabela ou exibição no SQL Server. É de suma importância a utilização da cláusula WHERE caso necessário.

DELETE FROM Nome tabela WHERE RE = '440545'

- **Comandos DCL - Data Control Language** os comandos que controlam o acesso aos dados.

-- Permissão de seleção na tabela Pessoa para a usuária Maria

GRANT SELECT ON Nome tabela TO Maria;

-- Permissão de seleção e inserção na tabela Pessoa para o usuário Maria

GRANT SELECT, INSERT ON Nome tabela TO Maria;

-- Negando alterações na tabela Pessoa para usuários do grupo GRP_RH

DENY INSERT, UPDATE, DELETE ON Nome tabela TO GRP_RH;

- **Comandos SQL – Pesquisa**

Para realizar uma pesquisa em um SGBD primeiramente devemos saber o que queremos, pois é recomendado utilizar o máximo possível de filtros para não sobrecarregar o sistema com pesquisas desnecessárias.

A estrutura da pesquisa se resume da seguinte forma.

- O que você deseja ver no banco de dados. - SELECT
- De qual banco e tabela ira vir as informações. - FROM
- Quais as especificações (filtros) você precisa. - WHERE

WHERE

A cláusula WHERE determina critérios de filtro e que somente as linhas que respeitem este critério sejam exibidas. A expressão contida no critério de filtro deve retornar TRUE ou FALSE. Utiliza-se dos seguintes operadores

Operadores relacionais:

Operador	Descrição
=	Compara se igual
<> ou !=	Compara se diferente
>	Compara se maior
<	Compara se menor
>=	Compara se maior ou igual
<=	Compara se menor ou igual

Operadores Lógicos:

Operador	Descrição
AND	Combina duas expressões e exige que sejam verdadeiras.
OR	Combina duas expressões e verifica se pelo menos uma das duas retorna verdadeira
NOT	Não retorna o conteúdo especificado

-- Traz o resultado cujo funcionário possua a o RE 440545

```
SELECT * FROM FUNCIONARIOS
```

```
WHERE RE = '440545'
```

-- Traz o resultado dos funcionários que possuam o salário igual ou maior que 1000

```
SELECT * FROM FUNCIONARIOS
```

```
WHERE SALARIO => '1000'
```

-- Traz o resultado cujo funcionário possua a o RE 440545 e o salário igual ou maior que 1000

```
SELECT * FROM FUNCIONARIOS
```

```
WHERE SALARIO = '1000' AND RE = '440545'
```

-- Traz todos os funcionários exceto os funcionários cujo salário é de 1000

```
SELECT * FROM FUNCIONARIOS  
  
WHERE NOT SALARIO = '1000'
```

BETWEEN:

Permite filtrar dados de uma consulta tendo como base uma faixa de valores, ou seja, um intervalo entre um valor menor e outro menor.

-- Traz todos os funcionários que possuam o salário entre 1000 e 2000

```
SELECT * FROM FUNCIONARIOS  
  
WHERE SALARIO BETWEEN '1000' AND '2000'
```

-- Traz todos os funcionários que possuam o salário fora da faixa de 1000 e 2000

```
SELECT * FROM FUNCIONARIOS  
  
WHERE SALARIO NOT BETWEEN '1000' AND '2000'
```

LIKE:

É utilizado para fazer pesquisas em dados do tipo texto. É útil quando não sabemos a forma exata do que queremos pesquisar. Por exemplo, sabemos que o nome da pessoa começa com Maria, mas não sabemos o restante do nome.

-- Traz como resultados todos os funcionários cujo primeiro nome seja Alberto

```
SELECT * FROM FUNCIONARIOS  
  
WHERE NOME LIKE 'ALBERTO%'
```

-- Traz como resultados todos os funcionários que não possuam o primeiro nome como Alberto

```
SELECT * FROM FUNCIONARIOS  
  
WHERE NOME NOT LIKE 'ALBERTO%'
```

IN:

O operador IN, pode ser utilizado no lugar do operador OR em determinadas situações, permite verificar se o valor de uma coluna está presente em uma lista de elementos.

-- Traz como resultado apenas os funcionários cujo RE 440510 e RE 440542

```
SELECT * FROM FUNCIONARIOS  
  
WHERE RE IN('440510', '440542')
```

NOT IN:

O operador NOT IN, por sua vez, ao contrário de IN permite obter como resultado o valor de uma coluna que não pertence a uma determinada lista de elementos.

-- Traz todos os funcionários exceto os funcionários cujo seja RE 440510 e RE 440542

```
SELECT * FROM FUNCIONARIOS  
  
WHERE RE NOT IN('440510', '440542')
```

IS NULL:

Quando um INSERT não faz referência a uma coluna existente em uma tabela, o conteúdo desta ficará nulo(NULL). Nota-se que este valor NÃO é um valor zero e nem mesmo uma string vazia.

-- Traz o resultado de todos os funcionários que possuam o salário entre 1000 e 2000 ou NÃO tenham o salário declarado na tabela.

```
SELECT * FROM FUNCIONARIOS  
  
WHERE SALARIO BETWEEN '1000' AND '2000' OR SALARIO IS NULL
```

ORDERBY:

Ele determina a ordem de apresentação do resultado de uma pesquisa de forma ascendente ou descendente.

-- Lista os dados da tabela com o campo RE em ordem ascendente

```
SELECT * FROM FUNCIONARIOS  
  
ORDER BY RE
```

-- Lista os dados da tabela com o campo RE em ordem descendente

```
SELECT * FROM FUNCIONARIOS  
  
ORDER BY RE DESC
```

TOP:

É utilizada para limitar o número de registros retornados por uma consulta e pode, por exemplo, garantir certo ganho de desempenho em algumas consultas que normalmente seriam compostas por uma quantidade muito grande de registros.

-- Lista os dez primeiros registros da tabela

`SELECT TO 10 * FROM FUNCIONARIOS`

Comandos SQL – Funções

LEN:

Retorna o número de caracteres da expressão da cadeia de caracteres especificada, excluindo espaços em branco à direita.

-- Retorna a quantidade de caracteres da sentença

`SELECT LEN ('MARIA E JOAO') AS NOME`

LEFT:

Retorna a parte da esquerda de uma cadeia de caracteres com o número de caracteres especificado.

-- Retorna os primeiros 5 caracteres da sentença

`SELECT LEFT ('MARIA E JOAO',5) AS NOME`

RIGHT:

Retorna a parte da direita de uma cadeia de caracteres com o número de caracteres especificado.

-- Retorna os 4 últimos caracteres da sentença

`SELECT RIGHT ('MARIA E JOAO',4) AS NOME`

REPLACE:

Substitui todas as ocorrências de um valor da cadeia de caracteres especificado por outro valor de cadeia de caracteres.

-- substitui todas as letras "S" por "X" na sentença

`SELECT REPLACE ('TESTO','S','X') AS RESULT`

REPLICATE:

Repete um valor da cadeia de caracteres um número especificado de vezes.

-- replica um caractere 0 quatro vezes na frente do valor 440542

```
SELECT REPLICATE ('0', 4) + '440542' AS RESULT
```

REVERSE:

Retorna a ordem inversa de um valor da cadeia de caracteres.

-- Retorna a sentença em ordem invertida

```
SELECT REVERSE ('BOM DIA A TODOS') AS RESULT
```

LTRIM:

Retorna uma expressão de caractere depois de remover espaços em branco à esquerda.

```
SELECT RTRIM ('Remover espaços a direita. ');
```

RTRIM:

Retorna uma cadeia de caracteres depois de truncar todos os espaços em branco à direita.

```
SELECT RTRIM ('    Remover espaços a esquerda.');
```

SPACE:

Retorna uma cadeia de caracteres de espaços repetidos.

-- Inclui 5 espaços entre os nomes

```
SELECT JOAO + SPACE (5) + 'SILVA'
```

STUFF:

A função STUFF insere uma cadeia de caracteres em outra cadeia de caracteres. Ela exclui um comprimento especificado de caracteres da primeira cadeia na posição inicial e, em seguida, insere a segunda cadeia na primeira, na posição inicial.

-- retorna uma cadeia de caracteres criada ao excluir três caracteres da primeira cadeia, abcdef, começando na posição 2, em b, e ao inserir a segunda cadeia de caracteres no ponto de exclusão.

```
SELECT STUFF ('abcdef', 2, 3, '000');
```

SUBSTRING:

É utilizada para obter uma parte dos dados armazenados.

-- Exibe o segundo, terceiro e quarto caracteres da constante de cadeia de caracteres 123456789

```
SELECT SUBSTRING ('123456789', 2, 3);
```

UPPER:

A função retorna uma expressão de caractere com dados de caracteres em minúsculas convertidos em maiúsculas.

-- Transforma para caracteres maiúsculo

```
select UPPER('letras para aumentar') AS MAIUSCULA
```

LOWER:

Retorna uma expressão de caractere depois de converter para minúsculas os dados de caracteres em maiúsculas.

-- Transforma para caracteres minúsculo

```
select LOWER('LETRAS A REDUZIR') AS MINUSCULA
```

- **Comandos SQL – Manipulação de datas**

Quando criamos colunas do tipo Data, podemos realizar uma série de cálculos e operações cronológicas. Podemos calcular números de dias entre duas datas, somar e subtrair dias, meses etc.

GETDATE ()

--Retorna a data e hora do sistema.

```
SELECT GETDATE () AS DATA_HORA
```

DAY ()

-- Retorna um valor inteiro que representa o dia da data especifica

```
SELECT DAY (GETDATE ()) AS DIA_ATUAL
```

MONTH ()

-- Retorna um valor inteiro que representa o mês da data especifica

```
SELECT MONTH (GETDATE ()) AS DIA_ATUAL
```

YEAR ()

Retorna um valor inteiro que representa o ano da data especifica

-- Retorna todos os empregados contratados no ano de 2000

```
SELECT * FROM FUNCIONARIOS
```

```
WHERE YEAR (DATAINCLUSAO) = 2000
```

DATEPART ()

É uma função do SQL Server que extrai uma parte específica do valor de data/hora.

-- Retorna o ano da data atual

```
SELECT DATEPART (YEAR, GETDATE());
```

Valor	Parte retornada	Abreviações
year	Ano	yy , yyyy
quarter	Trimestre (1/4 de ano)	qq , q
month	Mês	mm , m
dayofyear	Dia do ano	dy , y
day	Dia	dd , d
week	Semana	wk , ww
weekday	Dia da semana	dw
hour	Hora	hh
minute	Minuto	mi, n
second	Segundo	ss , s
millisecond	Milissegundo	ms
microsecond	Microssegundo	mcs
nanosecond	Nanossegundo	ns

DATENAME

Retorna uma cadeia de caracteres que representa uma porção da data.

-- Retorna o mês atual escrito por extenso

```
SELECT DATENAME (month, GETDATE ()) AS MES
```

-- Retorna o dia da semana atual escrito por extenso

```
SELECT DATENAME (weekday, GETDATE ()) AS DIA_SEMANA
```


DATEADD

Retorna uma data especificada com o intervalo number especificado (inteiro assinado) adicionado à datepart especificada dessa date.

-- Retorna a data de hoje mais 45 dias

```
SELECT DATEADD (DAY, 45, GETDATE())
```

-- Retorna a data de hoje mais 6 meses

```
SELECT DATEADD (MONTH, 6, GETDATE())
```

-- Retorna a data de hoje mais 2 ANOS

```
SELECT DATEADD (YEAR, 2, GETDATE())
```

DATEDIFF

Retorna a contagem (inteiro com sinal) dos limites especificados de datepart cruzados entre os parâmetros especificados startdate e enddate.

-- Conta a quantidade de dias que se passou desde 2000 até hoje

```
DECLARE @startdate datetime2 = '2000-01-01 12:10:09.3312722';
```

```
SELECT DATEDIFF (day, @startdate, GETDATE ());
```

Comandos SQL – Funções de grupo e agregação

Funções de grupo operam em conjuntos de linhas visando a fornecer um resultado para o grupo.

COUNT ()

Retorna o número de linhas que corresponde a um determinado critério.

-- Retorna o número de valores (valores nulos não serão contados) da coluna RE:

```
SELECT COUNT (RE) FROM FUNCIONARIOS
```

SUM ()

Retorna a soma total de uma coluna numérica.

-- Retorna a soma da coluna salario:

```
SELECT SUM(SALARIO) FROM FUNCIONARIOS
```

AVG ()

Calcula a média dos registros do campo informado.

-- Retorna a média da coluna salario:

```
SELECT AVG(SALARIO) FROM FUNCIONARIOS
```

MAX ()

Retorna o maior valor do campo especificado.

MIN ()

Retorna o menor valor do campo especificado.

-- Retorna o maior valor da coluna salario:

```
SELECT MAX(SALARIO) FROM FUNCIONARIOS
```

DISTINCT ()

Retorna somente valores distintos do campo especificado.

-- Retorna a lista de todos as ilhas distintas cadastradas

```
SELECT DISTINCT(PROGRAMA) AS PROGRAMAS FROM FUNCIONARIOS
```

GROUP BY

Agrupar linhas com base em valores de determinadas colunas. Desta forma, não estaremos trabalhando na pesquisa com todas as linhas da tabela, como fizemos anteriormente, mas sim em grupos menores.

-- Retorna a soma da coluna salario agrupando por ano de inclusão

```
SELECT YEAR (DATAINCLUSAO) AS ANO, SUM(SALARIO) AS SALARIO  
  
FROM FUNCIONARIOS  
  
GROUP BY YEAR (DATAINCLUSAO)
```

-- Conta quantos operadores cada supervisor possui

```
SELECT SUPERVISOR, COUNT (OPERADOR) AS OPERADOR  
  
FROM FUNCIONARIOS  
  
GROUP BY SUPERVISOR
```

-- Retorna a quantidade total de cada produto vendido

```
SELECT PRODUTO, COUNT (PRODUTO)  
  
FROM VENDAS  
  
GROUP BY PRODUTO
```

HAVING

A cláusula HAVING determina uma condição de busca para um grupo ou um conjunto de registros, definindo critérios para limitar os resultados obtidos a partir do agrupamento de registros. É importante lembrar que essa cláusula só pode ser usada em parceria com GROUP BY.

Obs.: O HAVING é diferente do WHERE. O WHERE restringe os resultados obtidos sempre após o uso da cláusula FROM, ao passo que a cláusula HAVING filtra o retorno do agrupamento.

-- Agrupa os resultados por tipo e fabricante, trazendo a quantidade de produtos em estoque

```
SELECT TIPO, FABRICANTE, SUM (QUANTIDADE) AS 'QUANTIDADE EM ESTOQUE'  
  
FROM PRODUTOS  
  
GROUP BY TIPO, FABRICANTE  
  
HAVING SUM(QUANTIDADE) > 200
```

- **Comandos SQL – Associando tabelas**

A associação entre tabelas, ou simplesmente JOIN entre tabelas, tem como principal objetivo trazer, em uma única consulta (SELECT), dados contidos em mais de uma tabela.

Normalmente esta associação é feita através de chave estrangeira de uma tabela com a chave primaria de outra. Mas isso não é um pré-requisito para o JOIN, de forma que qualquer informação comum entre duas tabelas servirá para associá-las.

Diferentes tipos de associação podem ser escritos com a ajuda das clausulas JOIN e WHERE. Por exemplo, podemos obter apenas os dados relacionados entre duas tabelas associadas. Também podemos combinar duas tabelas de forma que seus dados relacionados e não relacionados sejam obtidos.

INNER JOIN

Permite usar um operador de comparação para comparar os valores de colunas provenientes de tabelas associadas. Por meio desta cláusula, os registros de duas tabelas são usados para que sejam gerados os dados relacionados de ambas.

No exemplo abaixo, iremos criar as tabelas Cargo e Funcionário.

	IdCargo	NomeCargo
1	1	Programador Jr.
2	2	Web Designer Pl.
3	3	Programador Pl.
4	4	DBA Jr.
5	5	Programador Sr.

	IdFuncionario	IdCargo	NomeFuncionario	SalarioFuncionario
1	1	2	Tirica	2500.00
2	2	1	Zé da Pizza	2250.00
3	3	3	Tiozão do Gás	2750.00
4	4	4	Adalberto do Sacolão	2300.00
5	5	1	Marisa da Horta	2500.00

Repare que temos a coluna IdCargo nas duas tabelas, porém, ela possui finalidades distintas: enquanto na tabela Cargo, ela é chave primária, na tabela Funcionario ela é chave estrangeira.

Assim, a associação entre as tabelas é feita pela coluna IdCargo e podemos identificar os cargos existentes e o nome dos funcionários que desempenham cada um deles.

Usamos então a cláusula INNER JOIN para obtermos os dados relacionados das duas tabelas, para que sejam retornados todos os cargos ocupados pelos funcionários, bem como todos os funcionários que desempenham algum cargo. Veja como isso é feito no script abaixo:

-- Junta as tabelas funcionarios e cargo trazendo campos de ambas

```

SELECT C.NOMECargo [CARGO], F.NOMEFuncionario AS [FUNCIONÁRIO],
F.SALARIOFuncionario AS [SALÁRIO]

FROM CARGO AS C

INNER JOIN FUNCIONARIO AS F

ON C.IDCARGO = F.IDCARGO

```

É de pratica apelidar as tabelas para facilitar a escrita da consulta. (CARGO AS C e FUNCIONARIO AS F).

A sigla ON identifica a chave(campo) em comum em ambas tabelas que será utilizado para comparação

Nossa consulta terá o seguinte retorno:

	CARGO	FUNCIONÁRIO	SALÁRIO
1	Web Designer Pl.	Tirica	2500.00
2	Programador Jr.	Zé da Pizza	2250.00
3	Programador Pl.	Tiozão do Gás	2750.00
4	DBA Jr.	Adalberto do Sacolão	2300.00
5	Programador Jr.	Marisa da Horta	2500.00

LEFT JOIN

Permite obter não apenas os dados relacionados de duas tabelas, mais também os dados não relacionados encontrados na tabela à esquerda da cláusula JOIN. Caso não existam dados relacionados entre as tabelas à esquerda e a direita do JOIN, os valores resultantes de todas as colunas da lista de seleção da tabela à direita serão nulos.

Para exemplificar melhor, vejamos os exemplos das tabelas Cargo e Funcionário. Como dito anteriormente, o único cargo que não contém funcionário vinculado a ele é o Programador Sr. Para obtermos mesmo assim esse cargo, usamos a cláusula LEFT JOIN à esquerda do sinal de igual (=), como no script abaixo:

```
SELECT C.NOMECargo [CARGO], F.NOMEFuncionario AS [FUNCIONÁRIO],  
F.SALARIOFuncionario AS [SALÁRIO]  
  
FROM CARGO AS C  
  
LEFT JOIN FUNCIONARIO AS F  
  
ON C.IDCARGO = F.IDCARGO
```

Seu resultado será o seguinte:

CARGO	FUNCIONÁRIO	SALÁRIO
Programador Jr.	Zé da Pizza	2250.00
Programador Jr.	Marisa da Horta	2500.00
Web Designer Pl.	Tirica	2500.00
Programador Pl.	Tiozão do Gás	2750.00
DBA Jr.	Adalberto do Sacolão	2300.00
Programador Sr.	NULL	NULL

RIGHT JOIN

Ao contrário do LEFT JOIN, a cláusula RIGHT JOIN ou RIGHT OUTER JOIN retorna todos os dados encontrados na tabela à direita de JOIN. Caso não existam dados associados entre as tabelas à esquerda e à direita de JOIN, serão retornados valores nulos.

Suponhamos que a posição das tabelas usadas nos exemplos anteriores foi trocada. Se mesmo assim desejamos obter o mesmo resultado obtido anteriormente, podemos usar a cláusula RIGHT JOIN, assim

iremos conseguir tanto os dados relacionados como os não relacionados disponíveis na tabela à direita da cláusula JOIN.

```
SELECT C.NOMECargo [CARGO], F.NOMEFuncionario AS [FUNCIONÁRIO],  
F.SALARIOFuncionario AS [SALÁRIO]  
  
FROM Funcionario AS F  
  
RIGHT JOIN Cargo AS C  
  
ON F.IDCargo = C.IDCargo
```

Seu resultado será o seguinte:

CARGO	FUNCIONÁRIO	SALÁRIO
Programador Jr.	Zé da Pizza	2250.00
Programador Jr.	Marisa da Horta	2500.00
Web Designer Pl.	Tiririca	2500.00
Programador Pl.	Tiozão do Gás	2750.00
DBA Jr.	Adalberto do Sacolão	2300.00
Programador Sr.	NULL	NULL

FULL JOIN

Todas as linhas de dados da tabela à esquerda de JOIN e da tabela à direita serão retornadas pela cláusula FULL JOIN ou FULL OUTER JOIN. Caso uma linha de dados não esteja associada a qualquer linha da outra tabela, os valores das colunas a lista de seleção serão nulos. Caso contrário, os valores obtidos serão baseados nas tabelas usadas como referência.

```
SELECT C.NOMECargo [CARGO], F.NOMEFuncionario AS [FUNCIONÁRIO],  
F.SALARIOFuncionario AS [SALÁRIO]  
  
FROM Funcionario AS F  
  
FULL JOIN Cargo AS C  
  
ON F.IDCargo = C.IDCargo
```

Seu resultado será o seguinte:

CARGO	FUNCIONÁRIO	SALÁRIO
Web Designer Pl.	Tiririca	2500.00
Programador Jr.	Zé da Pizza	2250.00
Programador Pl.	Tiozão do Gás	2750.00
DBA Jr.	Adalberto do Sacolão	2300.00
Programador Jr.	Marisa da Horta	2500.00
Programador Sr.	NULL	NULL

CROSS JOIN

Todos os dados da tabela à esquerda de JOIN são cruzados com os dados da tabela à direita de JOIN por meio do CROSS JOIN, também conhecido como produto cartesiano. É possível cruzarmos informações de duas ou mais tabelas.

Para facilitar a compreensão a respeito desse tipo de associação, vamos usar as tabelas acima como exemplo. Caso a intenção seja exibir os dados de modo que todos os funcionários tenham todos os cargos e vice-versa. Para isso, devemos usar o CROSS JOIN, como no exemplo a seguir:

```
SELECT C.NOMECargo [CARGO], F.NOMEFuncionario AS [FUNCIONÁRIO],  
F.SALARIOFuncionario AS [SALÁRIO]  
  
FROM CARGO AS C  
  
CROSS JOIN FUNCIONARIO AS F  
  
ORDER BY C.NOMECargo
```

Seu resultado será o seguinte:

	CARGO	FUNCIONÁRIO	SALÁRIO
1	DBA Jr.	Tirica	2500.00
2	DBA Jr.	Zé da Pizza	2250.00
3	DBA Jr.	Tiozão do Gás	2750.00
4	DBA Jr.	Adalberto do Sacolão	2300.00
5	DBA Jr.	Marisa da Horta	2500.00
6	Programador Jr.	Tirica	2500.00
7	Programador Jr.	Zé da Pizza	2250.00
8	Programador Jr.	Tiozão do Gás	2750.00
9	Programador Jr.	Adalberto do Sacolão	2300.00
10	Programador Jr.	Marisa da Horta	2500.00
11	Programador Pl.	Tirica	2500.00
12	Programador Pl.	Zé da Pizza	2250.00
13	Programador Pl.	Tiozão do Gás	2750.00
14	Programador Pl.	Adalberto do Sacolão	2300.00
15	Programador Pl.	Marisa da Horta	2500.00
16	Programador Sr.	Tirica	2500.00
17	Programador Sr.	Zé da Pizza	2250.00
18	Programador Sr.	Tiozão do Gás	2750.00
19	Programador Sr.	Adalberto do Sacolão	2300.00
20	Programador Sr.	Marisa da Horta	2500.00
21	Web Designer Pl.	Tirica	2500.00
22	Web Designer Pl.	Zé da Pizza	2250.00
23	Web Designer Pl.	Tiozão do Gás	2750.00
24	Web Designer Pl.	Adalberto do Sacolão	2300.00
25	Web Designer Pl.	Marisa da Horta	2500.00

UNION

O UNION é uma cláusula responsável por unir informações obtidas a partir de diversos comandos SELECT. Para obtermos estes dados, não é obrigatório que as tabelas que as possuem estejam relacionadas.

Para exemplificar nossos conceitos, vamos considerar as tabelas Cargo, Clientes e Funcionarios, conforme a imagem abaixo.

	IdCargo	NomeCargo
1	1	Programador Jr.
2	2	Web Designer Pl.
3	3	Programador Pl.
4	4	DBA Jr.
5	5	Programador Sr.

	IdCliente	Nome
1	1	Carlos Souza
2	2	Maria Cristina
3	3	Eustáquio Quirino

	IdFuncionario	NomeFuncionario
1	1	Tirica
2	2	Zé da Pizza
3	3	Tiozão do Gás
4	4	Adalberto do Sacolão
5	5	Marisa da Horta

Supondo que precisamos obter o nome de todos os Cargos, Clientes e Funcionários das três tabelas acima, podemos usar o UNION ALL pra isso. Veja como no script a seguir:

```
SELECT IDCARGO AS 'ID', NOMECargo AS 'Cargo / Nome', 'Cargo' AS TABELA
FROM CARGO

UNION

SELECT IDCLIENTE, NOME, 'CLIENTES'
FROM CLIENTES

UNION

SELECT IDFUNCIONARIO, NOMEFUNCIONARIO, 'FUNCIONÁRIO'
FROM FUNCIONARIO

ORDER BY TABELA
```


	ID	CARGO / NOME	TABELA
1	1	Programador Jr.	CARGO
2	2	Web Designer Pl.	CARGO
3	3	Programador Pl.	CARGO
4	4	DBA Jr.	CARGO
5	5	Programador Sr.	CARGO
6	1	Carlos Souza	CLIENTES
7	2	Maria Cristina	CLIENTES
8	3	Eustáquio Quirino	CLIENTES
9	1	Tinica	FUNCIONÁRIO
10	2	Zé da Pizza	FUNCIONÁRIO
11	3	Tiozão do Gás	FUNCIONÁRIO
12	4	Adalberto do Sacolão	FUNCIONÁRIO
13	5	Marisa da Horta	FUNCIONÁRIO

• Comandos SQL – Consultas com subqueries

Uma consulta aninhada em uma instrução SELECT, INSERT, UPDATE ou DELETE é chamada de Subquery (subconsulta). O limite máximo de alinhamento de uma Subquery é de 32 níveis, limite que pode variar de acordo com a complexidade das outras instruções que compõem a consulta e também da quantidade de memória disponível. Confira as principais características das Subqueries:

A seguir os formatos normalmente apresentados pelas instruções que possuem uma Subquery:

- WHERE expressão [NOT] IN (Subquery);
- WHERE expressão operador_de_comparacao [ANY|ALL] (Subquery);
- WHERE [NOT] EXISTS (Subquery).

Subqueries usando IN/NOT

Observe as tabelas Cargo e Funcionario:

	Id	Cargo
1	4	DBA Jr.
2	1	Programador Jr.
3	3	Programador Pl.
4	5	Programador Sr.
5	2	Web Designer Pl.

	Id Funcionário	Id Cargo	Funcionário	Salário
1	1	2	Tinica	2750.00
2	2	1	Zé da Pizza	2250.00
3	3	3	Tiozão do Gás	3025.00
4	4	4	Adalberto do Sacolão	2530.00
5	5	1	Marisa da Horta	2750.00

Aqui temos um relacionamento entre as tabelas, onde cada funcionário tem seu cargo definido pelo Id da tabela Cargo. Perceba também que não temos nenhum funcionário com o cargo Programador Sr. referente ao Id 5. Isso é claro, porque a tabela Funcionario não contém este Id.

Caso precisemos informar ao usuário qual Cargo não está vinculado a algum Funcionário devemos usar a seguinte instrução:

```
SELECT IDCARGO AS Id, NOMECargo AS Cargo FROM CARGO  
WHERE IDCARGO NOT IN (SELECT IDCARGO FROM FUNCIONARIO)
```

O resultado será o seguinte:

	Id	Cargo
1	5	Programador Sr.

Ao executarmos o segundo SELECT do comando acima, serão obtidos os dados que, por sua vez, serão armazenados na memória. Assim que outro SELECT for executado, os dados armazenados devido à execução do SELECT anterior irão se confrontar, o que resultará na exibição da imagem acima, que é o cargo cujo código não está presente na tabela Funcionario, ou seja, o Id de cargo 5, referente ao Programador Sr.

Dito isto, podemos concluir que o SQL Server sempre executa o Select de dentro primeiro para depois executar o Select de fora.

Da mesma forma que no exemplo anterior, se usarmos a cláusula IN, teremos somente os Cargos vinculados aos Funcionários:

```
SELECT IDCARGO AS Id, NOMECargo AS Cargo FROM CARGO  
WHERE IDCARGO IN (SELECT IDCARGO FROM FUNCIONARIO)
```

	Id	Cargo
1	1	Programador Jr.
2	2	Web Designer Pl.
3	3	Programador Pl.
4	4	DBA Jr.

Subqueries introduzidas com o sinal de igual (=)

Observar a tabela:

	IdFuncionario	IdCargo	NomeFuncionario	SalarioFuncionario
1	1	2	Tirica	2750.00
2	2	1	Zé da Pizza	2250.00
3	3	3	Tiozão do Gás	3025.00
4	4	4	Adalberto do Sacolão	2530.00
5	5	1	Marisa da Horta	2750.00

na imagem acima que o Tiozão do Gás possui o maior salário (3025.00), enquanto que o Zé da Pizza possui o menor salário (2250.00). Conseguimos obter esses valores, mínimo e máximo, por meio das funções de totalização Min() e Max().

Com essas funções, podemos saber qual funcionário detém o maior salário e qual recebe o menor, como o script abaixo nos mostra usando o Max().

```
SELECT * FROM FUNCIONARIO  
  
WHERE SALARIOFUNCIONARIO = (SELECT MAX(SALARIOFUNCIONARIO)  
  
FROM FUNCIONARIO)
```

O resultado será o seguinte:

	IdFuncionario	IdCargo	NomeFuncionario	SalarioFuncionario
1	3	3	Tiozão do Gás	3025.00

Assim que a Subquery da instrução acima for executada, o maior salário será obtido e armazenado na memória. Tendo esse dado, o primeiro SELECT irá relacionar somente o funcionário que detém este valor. Lembre-se: a ordem de execução do SQL Server (neste caso específico) é sempre de dentro pra fora.

Update com Subqueries

As Subqueries podem ser usadas para atualizarmos dados. Por exemplo: Se quisermos dar um aumento de 10% somente aos funcionários que seus cargos forem de **Programador Jr.** podemos usar a seguinte instrução:

```
UPDATE FUNCIONARIO  
  
SET SALARIOFUNCIONARIO = SALARIOFUNCIONARIO * 1.1  
  
WHERE IDCARGO = (SELECT IDCARGO  
  
FROM CARGO  
  
WHERE IDCARGO = 1)
```

A com os salários atualizados ficaria desta forma:

	IdFuncionario	IdCargo	NomeFuncionario	SalarioFuncionario
1	1	2	Tirica	2750.00
2	2	1	Zé da Pizza	2475.00
3	3	3	Tiozão do Gás	3025.00
4	4	4	Adalberto do Sacolão	2530.00
5	5	1	Marisa da Horta	3025.00

Perceba que foram atualizados os salários apenas dos funcionários **Zé da Pizza** e **Marisa da Horta**.

Delete com Subqueries

Podemos usar também Subqueries para remover dados de uma tabela. Seguindo a lógica do exemplo anterior, vamos excluir somente os funcionários que tiverem o cargo DBA Jr. Conseguimos isso usando a seguinte instrução:

```
DELETE FROM FUNCIONARIO
```

```
WHERE IDCARGO = (SELECT IDCARGO
```

```
FROM CARGO
```

```
WHERE IDCARGO = 4)
```

	IdFuncionario	IdCargo	NomeFuncionario	SalarioFuncionario
1	1	2	Tirica	2750.00
2	2	1	Zé da Pizza	2475.00
3	3	3	Tiozão do Gás	3025.00
4	5	1	Marisa da Horta	3025.00

Note que o Adalberto do Sacolão foi excluído da base.

Subconsultas com EXISTS

Quando uma subconsulta é apresentada com a palavra-chave EXISTS, a subconsulta funciona como um teste de existência. A cláusula WHERE da consulta externa testa se as linhas retornadas pela subconsulta existem. A subconsulta não produz de fato nenhum dado; ela retorna um valor TRUE ou FALSE.

Observe as tabelas:

ALIMENTOS		FRUTAS	
<i>id_SERIAL</i>	<i>nome</i>	<i>id_SERIAL</i>	<i>nome</i>
NULL	maçã	NULL	banana
NULL	espinafre	NULL	maçã
NULL	manga	NULL	manga
NULL	alface	NULL	goiaba

Neste exemplo a tabela frutas é consultada para verificar se o alimento é uma fruta ou não. Caso o alimento conste da tabela frutas é uma fruta, caso não conste não é uma fruta. Abaixo está mostrado o script utilizado para criar e carregar as tabelas e executar a consulta.

```
SELECT nome, CASE WHEN EXISTS (SELECT nome FROM #frutas WHERE nome=a.nome)
    THEN 'sim'
    ELSE 'não'
END AS fruta
FROM #alimentos a
```

Resultado:

<i>nome</i>	<i>fruta</i>
maçã	sim
espinafre	não
manga	sim
alface	não

- **Comandos SQL – Consultas avançadas**

CASE

Avalia uma lista de condições e retorna uma das várias expressões de resultado possíveis.

CASE pode ser usada em qualquer instrução ou cláusula que permita uma expressão válida. Por exemplo, você pode usar CASE em instruções, como SELECT, UPDATE, DELETE e SET, e em cláusulas, como select_list, IN, WHERE, ORDER BY e HAVING.

Suponha que exista em sua aplicação, a tabela PEDIDOS. Esta tabela possui os seguintes campos: ID, DATA da compra, forma de pagamento e o valor. O campo forma de pagamento (FORMA_PAGAMENTO) pode assumir os seguintes valores: 1 (boleto bancário), 2 (cartão de crédito VISA) ou 3 (cartão de crédito MASTERCARD).

ID	DATA	FORMA_PAGAMENTO	VALOR
1	39791	1	108
2	39708	1	52
2	39709	1	328

Se quiséssemos mostrar o nome da forma de pagamento escolhida, poderíamos fazer da seguinte forma:

SELECT DATA,

CASE id

WHEN 1 THEN 'Boleto Bancário'

WHEN 2 THEN 'Cartão de Crédito VISA'

WHEN 3 THEN 'Cartão de Crédito MASTERCARD'

END as PG,

VALOR

FROM #PEDIDOS

Resultado:

DATA	PG	VALOR
39791	Boleto Bancário	108
39708	Cartão de Crédito VISA	52
39709	Cartão de Crédito VISA	328

COALESCE

Esta função é utilizada quando um conjunto de campos for passado e deve ser retornado o primeiro não nulo, esta função é muito útil quando se quer fazer uma soma de campos numéricos e um deles é nulo, com esta função podemos retornar zero quando campo for nulo.

Observe a seguinte tabela:

ID_CLIENTE	nome	razao_social	CPJ_CNPJ	Ativo	Saldo
1	PADARIA DOCES E SABORES	PADARIA DS	1234567890	S	-100.00
2	PEDRO DA SILVA TADEU	<NULL>	8415784548	N	1200.00
3	JOAO JOSE FARINHA	<NULL>	3232242342	N	-10.00
4	FLAVIA MARIA PEDROSCA	FMP	1294316576	S	2554.00
5	DEVMEDIA	CLUBE DELPHI	123456789	S	<NULL>

O exemplo abaixo retorna a coluna com o nome que não for nulo entre o campo NOME e RAZAO_SOCIAL já a coluna SALDO irá retornar zero quando a coluna for nulo.

SELECT

ID_CLIENTE, COALESCE (NOME, RAZAO_SOCIAL) as NOME_APRESENTACAO,

NOME, RAZAO_SOCIAL, COALESCE (SALDO, 0.0) as SALDO

FROM #CLIENTE

Resultado:

ID_CLIENTE	NOME_APRESENTACAO	NOME	RAZAO_SOCIAL	SALDO
1	PADARIA DS	PADARIA DOCES E SABORES	PADARIA DS	-100.00
2	PEDRO DA SILVA TADEU	PEDRO DA SILVA TADEU	NULL	1200.00
3	JOAO JOSE FARINHA	JOAO JOSE FARINHA	NULL	-10.00
4	FMP	FLAVIA MARIA PEDROSCA	FMP	2554.00
5	CLUBE DELPHI	DEVMEDIA	CLUBE DELPHI	0.00

NULIF

Esta função é utilizada para comparação de dois parâmetros, sendo os dois iguais o retorno será null, caso contrário o primeiro parâmetro será retornado.

O exemplo abaixo retorna null quando a coluna saldo tiver o saldo igual a -10.

```
SELECT ID_CLIENTE, NOME, saldo, NULLIF(SALDO, -10) as SALDO_nullif  
  
FROM CLIENTE
```

ID_CLIENTE	NOME	saldo	SALDO_nullif
1	PADARIA DOCES E SABORES	-100.00	-100.00
2	PEDRO DA SILVA TADEU	1200.00	1200.00
3	JOAO JOSE FARINHA	-10.00	NULL
4	FLAVIA MARIA PEDROSCA	2554.00	2554.00
5	DEVMEDIA	NULL	NULL

- **Comandos SQL – Programando no SQL Server**

O uso da programação no SQL Server envolve a criação e uso de variáveis, o uso de elementos de controle de fluxo e também o uso de Stored Procedures. Assim conseguimos incluir uma determinada lógica em nossas queries.

Variáveis de Usuário

Uma variável local do Transact-SQL é um objeto nos scripts e batches que mantém um valor de dado. Através do comando DECLARE, podemos criar variáveis locais, sendo feito isso no corpo de uma procedure ou batch.

Podemos ver abaixo, dois exemplos de declarações de variáveis, ambos estão corretos, depende do programador qual irá utilizar.

--Exemplo 1

```
DECLARE @IdUsuario AS INT, @Nome AS VARCHAR(50), @Idade AS SMALLINT
```

--Exemplo 2

```
DECLARE @IdUsuario AS INT
```

```
DECLARE @Nome AS VARCHAR(50)
```

```
DECLARE @Idade AS SMALLINT
```

Em ambos os exemplos temos o uso do caractere coringa @ antes da variável. Este é um padrão para declarar variáveis e deve ser seguido pelos programadores. Podemos notar ainda que cada variável declarada tem seu datatype atribuído.

É possível atribuir valores para cada uma das variáveis. Para isso, usamos o comando SET ou o SELECT, como vemos no exemplo.

--Exemplo 1

```
SET @Nome = 1
```

--Exemplo 2

```
SELECT @Nome = ColunaNome FROM Tabela @Idade = ColunaIdade FROM Tabela
```

Seja com SET ou SELECT, conseguimos atribuir valor as variáveis, depende da necessidade de cada desenvolvedor.

Controle de Fluxo

Assim como muitas linguagens de programação utilizam operadores de condição, o SQL não poderia ficar de fora. Ele trabalha com esses elementos, também denominado de controle de fluxo, permitindo assim ao desenvolvedor criar lógicas para as mais variadas situações e regras de negócio de seu sistema.

IF/ELSE

Impõe condições na execução de uma instrução Transact-SQL. A instrução Transact-SQL que segue uma palavra-chave IF e sua condição será executada se a condição for satisfeita: a expressão Booleana retorna TRUE. A palavra-chave opcional ELSE introduz outra instrução Transact-SQL que será executada quando a condição IF não for atendida: a expressão Booleana retorna FALSE.

```
DECLARE @VARIABEL AS INT
```

```
SET @VARIABEL = 15
```

```
IF @VARIABEL >=10
```

```
    BEGIN
```

```
        SELECT 'sua variável é maior ou igual que 10'
```

```
    END
```

```
ELSE
```

```
    BEGIN
```

```
        SELECT 'sua variável é menor que 10'
```

```
    END
```

WHILE

Faz com que um comando ou bloco de comandos SQL seja executado repetidamente, ou seja, é criado um loop o comando ou bloco de comandos, que será executado enquanto a condição especificada for verdadeira.

```
CREATE TABLE #CALENDARIO (DATAS DATE, TIPO VARCHAR(50)); /* CRIAR TABELA CALENDARIO */
```

```
/* ALIMENTAR TABELA TEMPORARIA COM O CALENDARIO DEFINIDO NAS VARIAVEIS -- */
```

```
DECLARE @DT_INICIO DATE, @DT_FIM DATE;
```

```
SET @DT_INICIO = '2014-11-21';
```

```
SET @DT_FIM = '2014-12-20';
```

```

WHILE (@DT_INICIO <= @DT_FIM)

BEGIN

    INSERT INTO #CALENDARIO (DATAS, TIPO)

        VALUES (@DT_INICIO,

            CASE

                WHEN DATEPART(WEEKDAY, @DT_INICIO) BETWEEN '2' AND '6' THEN 'UTIL'

                WHEN DATEPART(WEEKDAY, @DT_INICIO) = '1' THEN 'DOM'

                WHEN DATEPART(WEEKDAY, @DT_INICIO) = '7' THEN 'SAB'

            END)

        SET @DT_INICIO = DATEADD(DAY, 1, @DT_INICIO)

END

```

Este exemplo utiliza as variáveis @DT_INICIO e @DT_FIM para obter um período. O comando WHILE utiliza-se da instrução SELECT INTO para ir preenchendo a tabela com a data e o TIPO (dia da semana conforme comando CASE) até a @DT_INICIO ser menor ou igual a @DT_FIM.

DATAS	TIPO
27/03/2015	UTIL
28/03/2015	SAB
29/03/2015	DOM
30/03/2015	UTIL
31/03/2015	UTIL
01/04/2015	UTIL
02/04/2015	UTIL

CURSOR

Cursores são mecanismos que permitem que as linhas de uma tabela sejam manipuladas uma a uma. Atuam como ponteiros que apontam para as linhas que formam o resultado de uma dada consulta que é armazenada no cursores. Podemos recuperar e manipular os valores de cada linha apontada por um cursores.

Veja abaixo a sintaxe para criação de um cursor no SQL Server utilizando a tabela anterior.

```
DECLARE @DT VARCHAR(30) -- Declara variavel @DT
```

```
DECLARE @TP VARCHAR(30)
```

```
DECLARE BD_CURSOR CURSOR FOR -- Declara CURSOR
```

```
SELECT DATAS, TIPO FROM #CALENDARIO -- Consulta para alimentar o cursor
```

```
OPEN BD_CURSOR -- Abre o CURSOR (OPEN)
```

```
FETCH NEXT FROM BD_CURSOR INTO @DT, @TP -- Inclui as informações da primeira linha do cursor nas variáveis @DT e @TP.
```

```
WHILE @@FETCH_STATUS = 0 -- Enquanto não terminam as linhas dentro do cursor, faça.
```

```
BEGIN
```

```
PRINT 'BOM DIA HOJE É ' + @DT + (CASE WHEN @TP = 'UTIL' THEN ' - DIA DE TRABALHO' ELSE ' - DIA DE FOLGA' END)
```

```
FETCH NEXT FROM BD_CURSOR INTO @DT, @TP -- pula para a próxima linha do cursor
```

```
END
```

```
-- fecha o cursor
```

```
CLOSE BD_CURSOR
```

```
-- desloca o cursor liberando a memoria
```

```
DEALLOCATE BD_CURSOR
```

Resultado:

```
BOM DIA HOJE É 2015-03-27 - DIA DE TRABALHO
```

```
BOM DIA HOJE É 2015-03-28 - DIA DE FOLGA
```

```
BOM DIA HOJE É 2015-03-29 - DIA DE FOLGA
```

```
BOM DIA HOJE É 2015-03-30 - DIA DE TRABALHO
```

```
BOM DIA HOJE É 2015-03-31 - DIA DE TRABALHO
```

```
BOM DIA HOJE É 2015-04-01 - DIA DE TRABALHO
```

BOM DIA HOJE É 2015-04-02 - DIA DE TRABALHO

TRIGGER

Um gatilho é um tipo especial de procedimento armazenado que é executado automaticamente quando um evento ocorre no servidor de banco de dados.

Para entender uma TRIGGER primeiramente temos que entender como funciona o fluxo de transações no SQL SERVER

Fluxo de transações no SQL SERVER

Todas as transações DML (INSERT, UPDATE OU DELETE) no banco de dados seguem um fluxo de transações que não vemos, mas que o gerenciados do banco de dados executa automaticamente:

- Verificações de IDENTITY INSERT – Verifica se o dado é auto inserido
- Restrições (CONSTRAINT) de nulos (NULL) – Verificação se há restrições de nulos
- Checagem do tipo de dados - Analisa se o dado inserido é compatível com o tipo de dados definido na criação da tabela (varchar, int, etc)
- Execução da trigger INSTEAD OF – A execução da instrução DML para por aqui e é executada a TRIGGER.
- Restrição de Chave Primária -
- Restrição de Check
- Restrição de Chave Estrangeira
- Execução da DML e atualização do log de transação
- Execução da trigger AFTER
- Commit da transação – Confirmação

Os **gatilhos DML** são executados quando um usuário tenta modificar dados através de um evento DML (linguagem de manipulação de dados). Os eventos DML são instruções INSERT, UPDATE ou DELETE em uma tabela ou exibição. Esses gatilhos são disparados quando qualquer evento válido é acionado, independentemente de quaisquer linhas da tabela serem afetadas ou não.

CREATE TRIGGER TRIGGER_NAME

ON NOME_DA_TABLE | NOME_DA_VIEW

FOR | AFTER | INSTEAD OF

INSERT, UPDATE, DELETE

AS

--Bloco contendo as instruções que serão executadas na TRIGGER

Os **gatilhos DDL** são executados em resposta a diversos eventos DDL (linguagem de definição de dados). Esses eventos correspondem, basicamente, a instruções Transact-SQL CREATE, ALTER e DROP e determinados procedimentos armazenados do sistema que executam operações do tipo DDL.

CREATE TRIGGER TRIGGER_NAME

ON ALL SERVER | DATABASE

FOR | AFTER EVENT_TYPE | EVENT_GROUP

AS

--Bloco contendo as instruções que serão executadas na TRIGGER

Os **gatilhos de logon** são disparados em resposta ao evento LOGON que é gerado quando as sessões de um usuário estão sendo estabelecidas.

CREATE TRIGGER TRIGGER_NAME

ON ALL SERVER

FOR| AFTER LOGON

AS SQL_STATEMENT

--Bloco contendo as instruções que serão executadas na TRIGGER

Tipos de TRIGGER

AFTER

- A trigger é executada após a confirmação de uma transação no banco de dados (INSERT, UPDATE OU DELETE)
- Se não declarado o tipo, o AFTER é declarado como padrão
- AFTER pode só pode ser vinculada a TABELAS

INSTEAD OF

- O código presente na TRIGGER é executado no lugar da operação que causou o disparo, ou seja, as declarações DML (INSERT, UPDATE OU DELETE) não simuladas e não executadas.
- AFTER pode ser aplicado em VIEWS

COMO ENCONTRAR UMA TRIGGER PELO NOME

- Em uma tabela específica

EXEC SP_HELPTRIGGER @TABNAME=NOME_DA_TABELA

- No banco de dados todo

USE NOME_DO_BANCO_DE_DADOS

*SELECT **

FROM SYS.TRIGGERS

WHERE IS_DISABLED = 0

Está desabilitado?

0 não

1 sim

OVER ()

A cláusula OVER fornece cálculo cumulativos baseados nas linhas de um relatório e seu comportamento é muito diferente de tudo que estamos acostumados a usar. Entenda esta diferença a partir de agora.

As funções de agregação usadas com a cláusula GROUP BY, por exemplo, fazem cálculos sobre os valores de uma massa de dados extraída da base de dados. Assim, podemos calcular somas, médias e contagens sobre toda a massa de dados que atende aos critérios definidos na cláusula WHERE e depois agrupamos os resultados conforme os critérios que escrevemos na cláusula GROUP BY. Depois disso, teremos finalmente uma listagem dos resultados.

A cláusula OVER, por sua vez, só é executada depois que esta listagem estiver pronta. Se sua declaração SQL incluem filtros de dados (WHERE), agrupamentos (GROUP BY) e filtros de dados agrupados (HAVING), tudo isso vai ser calculado antes de rodar o OVER. E ao usarmos esta cláusula, estamos executando um cálculo especial que define uma nova coluna nesta listagem. A propósito, podemos definir quantas novas colunas quisermos, mas cada uma delas usará um cálculo com OVER.

Observe a seguinte tabela:

<i>Vendedor</i>	<i>Produto</i>	<i>Quantidade</i>	<i>Telefonemas</i>
João	Geladeira	5	15
Jonas	Geladeira	2	13
Rafael	Televisão	6	9
Rafael	Geladeira	1	6
João	Televisão	2	8
Jonas	Bicicleta	7	11
Rafael	Bicicleta	3	9
Jonas	Roupeiro	4	9
João	Computador	9	25
Rafael	Computador	1	5
João	Fogão	3	6
Rafael	Fogão	6	8
Jonas	Cama	2	3
João	Cama	5	9

Caso queiramos mostrar na mesma consulta o total de vendas por vendedor, o total de vendas da amostra, o percentual de vendas por produto e o percentual de vendas de produto por vendedor.

Caso focemos criar uma subconsulta para cada indicador, nossa consulta ficaria muito lenta e onerosa ao servidor.

Observe a seguinte consulta:

```
SELECT VENDEDOR, PRODUTO, QUANTIDADE, TELEFONEMAS,  
  
    SUM(QUANTIDADE) OVER (PARTITION BY VENDEDOR) AS VENDAS_VENDEDOR,  
  
    SUM(QUANTIDADE) OVER () AS VENDAS_TOTAS,  
  
    CAST (1. * QUANTIDADE / SUM(QUANTIDADE) OVER ()* 100 AS DECIMAL(5,2)) AS  
PERC_TOTAL,
```



```

CAST (1. * QUANTIDADE / SUM(QUANTIDADE) OVER (PARTITION BY VENDEDOR) * 100 AS
DECIMAL(5,2)) AS PERC_INDIVIDUAL
FROM #ANALISE_VENDAS
ORDER BY VENDEDOR, PRODUTO

```

Obtemos o seguinte resultado:

VENDEDOR	PRODUTO	QUANTIDADE	TELEFONEMAS	VENDAS_VENDEDOR	VENDAS_TOTAS	PERC_TOTAL	PERC_INDIVIDUAL
João	Cama	5	9	24	56	8.93	20.83
João	Computador	9	25	24	56	16.07	37.50
João	Fogão	3	6	24	56	5.36	12.50
João	Geladeira	5	15	24	56	8.93	20.83
João	Televisão	2	8	24	56	3.57	8.33
Jonas	Bicicleta	7	11	15	56	12.50	46.67
Jonas	Cama	2	3	15	56	3.57	13.33
Jonas	Geladeira	2	13	15	56	3.57	13.33
Jonas	Roupeiro	4	9	15	56	7.14	26.67
Rafael	Bicicleta	3	9	17	56	5.36	17.65
Rafael	Computador	1	5	17	56	1.79	5.88
Rafael	Fogão	6	8	17	56	10.71	35.29
Rafael	Geladeira	1	6	17	56	1.79	5.88
Rafael	Televisão	6	9	17	56	10.71	35.29

A coluna [VENDAS_VENDEDOR] somamos a quantidade “SUM(QUANTIDADE)” de cada vendedor OVER (PARTITION BY VENDEDOR).

A coluna [VENDAS_TOTAS] somamos a quantidade “SUM(QUANTIDADE)” de toda a amostra SUM(QUANTIDADE) OVER ().

A coluna [PERC_TOTAL] pegamos a quantidade dividida pela soma da quantidade total SUM(QUANTIDADE) OVER (), o restante da cláusula é para formatar o número.

A coluna [PERC_INDIVIDUAL] pegamos a quantidade de vendas dividida pela soma da quantidade total de vendas por vendedor SUM(QUANTIDADE) OVER (PARTITION BY VENDEDOR), o restante da cláusula é para formatar o número.

Funções para Ranking

Veremos quatro novas functions que foram criadas para nos ajudar a criar queries para análise de dados mais eficientes e com mais facilidade.

Utilizaremos como exemplo uma tabela que conterá as vendas de telemarketing por vendedor e tipo de produto, além da quantidade vendida. Um exemplo bem simples mas que poderemos utilizar para rankear os vendedores de várias maneiras.

<i>Vendedor</i>	<i>Produto</i>	<i>Quantidade</i>	<i>Telefonemas</i>
João	Geladeira	50000	15
Jonas	Geladeira	20000	13
Rafael	Televisão	60000	9
Rafael	Geladeira	10000	6
João	Televisão	20000	8
Jonas	Bicicleta	70000	11
Rafael	Bicicleta	30000	9
Jonas	Roupeiro	40000	9
João	Computador	90000	25
Rafael	Computador	10000	5
João	Fogão	30000	6
Rafael	Fogão	60000	8
Jonas	Cama	20000	3
João	Cama	50000	9

ROW_NUMBER

Esta função irá nos prover um número sequencial e inteiro para cada linha que o nossa query possuir.

```
SELECT ROW_NUMBER () OVER ( ORDER BY QUANTIDADE DESC ) AS NUMERO, VENDEDOR,
PRODUTO, QUANTIDADE, TELEFONEMAS
```

```
FROM ANALISE_VENDAS
```

Resultado:

<i>Numero</i>	<i>Vendedor</i>	<i>Produto</i>	<i>Quantidade</i>	<i>Telefonemas</i>
1	João	Computador	90000	25
2	Jonas	Bicicleta	70000	11
3	Rafael	Televisão	60000	9
4	Rafael	Fogão	60000	8

5	João	Cama	50000	9
6	João	Geladeira	50000	15
7	Jonas	Roupeiro	40000	9
8	Rafael	Bicicleta	30000	9
9	João	Fogão	30000	6
10	Jonas	Cama	20000	3
11	João	Televisão	20000	8
12	Jonas	Geladeira	20000	13
13	Rafael	Geladeira	10000	6
14	Rafael	Computador	10000	5

Podemos ainda acrescentar na nossa query a outra opção da sintaxe básica que se chama PARTITION BY, que funciona mais ou menos como o GROUP BY, definindo por qual critério nosso ROW_NUMBER deverá ser ressetado.

```
SELECT ROW_NUMBER () OVER (PARTITION BY PRODUTO ORDER BY QUANTIDADE DESC) AS
NUMERO, VENDEDOR, PRODUTO, QUANTIDADE

FROM ANALISE_VENDAS
```

Resultado:

NUMERO	VENDEDOR	PRODUTO	QUANTIDADE
1	Jonas	Bicicleta	70000
2	Rafael	Bicicleta	30000
1	João	Cama	50000
2	Jonas	Cama	20000
1	João	Computador	90000
2	Rafael	Computador	10000
1	Rafael	Fogão	60000
2	João	Fogão	30000
1	João	Geladeira	50000

2	Jonas	Geladeira	20000
3	Rafael	Geladeira	10000
1	Jonas	Roupeiro	40000
1	Rafael	Televisão	60000
2	João	Televisão	20000

Podemos notar que a diferença para nosso select anterior é que agora nossa coluna Número reinicia a contagem a cada novo produto além de ordenar por Quantidade.

RANK () e DENSE_RANK ()

O funcionamento destas funções é muito parecido com o ROW_NUMBER, elas também irão nos prover com um número sequencial e inteiro. A diferença principal é que RANK e DENSE_RANK atribuem o mesmo valor para as colunas que possuem o mesmo valor dentro da ordem que foi estabelecida.

Verificando o exemplo abaixo ficará mais claro:

```
SELECT VENDEDOR, PRODUTO, QUANTIDADE, ROW_NUMBER () OVER (ORDER BY QUANTIDADE
DESC) AS NUMERO,
```

```
RANK () OVER (ORDER BY QUANTIDADE DESC) AS RANK, DENSE_RANK () OVER (ORDER BY
QUANTIDADE DESC) AS DENSERANK
```

```
FROM ANALISE_VENDAS
```

```
ORDER BY QUANTIDADE DESC
```

Resultado:

Vendedor	Produto	Quantidade	Numero	Rank	DenseRank
João	Computador	90000	1	1	1
Jonas	Bicicleta	70000	2	2	2
Rafael	Televisão	60000	3	3	3
Rafael	Fogão	60000	4	3	3
João	Cama	50000	5	5	4
João	Geladeira	50000	6	5	4
Jonas	Roupeiro	40000	7	7	5

Rafael	Bicicleta	30000	8	8	6
João	Fogão	30000	9	8	6
Jonas	Cama	20000	10	10	7
João	Televisão	20000	11	10	7
Jonas	Geladeira	20000	12	10	7
Rafael	Geladeira	10000	13	13	8
Rafael	Computador	10000	14	13	8

Vamos então analisar os resultados, a função RANK () nos dará o número sequencial, mas irá repetir o valor para os registros que possuírem mesmo valor para a coluna que estamos analisando e pulando a sequência, já a função DENSE_RANK () também irá repetir o sequencial para as colunas que tiverem o mesmo valor, mas seguirá o sequencial na ordem correta.

NTILE

Esta função nos permite separar o resultado de uma query em um determinado número de grupos de acordo com uma ordem.

Se o número de linhas em uma partição não for divisível por inteiro, isso causará grupos de dois tamanhos que diferem por um membro. Grupos maiores aparecem antes de grupos menores na ordem especificada pela cláusula OVER. Por exemplo, se o número total de linhas for 53 e o número de grupos for cinco, os três primeiros grupos terão 11 linhas e os dois grupos restantes terão 10 linhas cada. Por outro lado, se o número total de linhas for divisível pelo número de grupos, as linhas serão igualmente distribuídas entre os grupos. Por exemplo, se o número total de linhas for 50 e houver cinco grupos, cada bucket conterá 10 linhas.

```
SELECT VENDEDOR, PRODUTO, QUANTIDADE, ROW_NUMBER () OVER (ORDER BY QUANTIDADE
DESC) AS NUMERO, NTILE(4) OVER(ORDER BY QUANTIDADE DESC) AS GRUPO

FROM ANALISE_VENDAS

ORDER BY QUANTIDADE DESC
```

Resultado:

VENDEDOR	PRODUTO	QUANTIDADE	NUMERO	GRUPO
João	Computador	90000	1	1
Jonas	Bicicleta	70000	2	1
Rafael	Televisão	60000	3	1

Rafael	Fogão	60000	4	1
João	Cama	50000	5	2
João	Geladeira	50000	6	2
Jonas	Roupeiro	40000	7	2
Rafael	Bicicleta	30000	8	2
João	Fogão	30000	9	3
Jonas	Cama	20000	10	3
João	Televisão	20000	11	3
Jonas	Geladeira	20000	12	4
Rafael	Geladeira	10000	13	4
Rafael	Computador	10000	14	4

Como podemos ver na coluna grupo, separamos os registros em quatro grupos, utilizando a seguinte fórmula
NumeroDeRegistros / NumeroDeGrupos.

Bibliografia:

Apostila Impacta SQL

<http://www.linhadecodigo.com.br/>

<http://www.devmedia.com.br/>

<https://programandodotnet.wordpress.com>

<https://technet.microsoft.com>

<http://pgdocptbr.sourceforge.net/>

<http://www.portaleducacao.com.br/>

<http://www.fabiobmed.com.br/>

Qualquer dúvida estou à disposição.

Alberto Tatuya Arizono Silva Filho

alberto.filho@atento.com