

Olá,

Criei estas apostilas a mais de 5 anos e atualizei uma série delas com alguns dados adicionais. Muitas partes desta apostila está desatualizada, mas servirá para quem quer tirar uma dúvida ou aprender sobre .Net e as outras tecnologias.

Perfil Microsoft: <https://www.mcpvirtualbusinesscard.com/VBCServer/msincic/profile>

Marcelo Sincic trabalha com informática desde 1988. Durante anos trabalhou com desenvolvimento (iniciando com Dbase III e Clipper S'87) e com redes (Novell 2.0 e Lantastic).

Hoje atua como consultor e instrutor para diversos parceiros e clientes Microsoft.

Recebeu em abril de 2009 o prêmio **Latin American MCT Awards** no MCT Summit 2009, um prêmio entregue a apenas 5 instrutores de toda a América Latina (<http://www.marcelosincic.eti.br/Blog/post/Microsoft-MCT-Awards-America-Latina.aspx>).

Recebeu em setembro de 2009 o prêmio **IT HERO** da equipe Microsoft Technet Brasil em reconhecimento a projeto desenvolvido (<http://www.marcelosincic.eti.br/Blog/post/IT-Hero-Microsoft-TechNet.aspx>). Em Novembro de 2009 recebeu novamente um premio do programa IT Hero agora na categoria de especialistas (<http://www.marcelosincic.eti.br/Blog/post/TechNet-IT-Hero-Especialista-Selecionado-o-nosso-projeto-de-OCS-2007.aspx>).

Acumula por 5 vezes certificações com o título **Charter Member**, indicando estar entre os primeiros do mundo a se certificarem profissionalmente em Windows 2008 e Windows 7.

Possui diversas certificações oficiais de TI:

- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2008
- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2005
- MCITP - Microsoft Certified IT Professional Windows Server 2008 Admin
- MCITP - Microsoft Certified IT Professional Enterprise Administrator Windows 7 Charter Member
- MCITP - Microsoft Certified IT Professional Enterprise Support Technical
- MCPD - Microsoft Certified Professional Developer: Web Applications
- MCTS - Microsoft Certified Technology Specialist: Windows 7 Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows Mobile 6. Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Active Directory Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Networking Charter Member
- MCTS - Microsoft Certified Technology Specialist: System Center Configuration Manager
- MCTS - Microsoft Certified Technology Specialist: System Center Operations Manager
- MCTS - Microsoft Certified Technology Specialist: Exchange 2007
- MCTS - Microsoft Certified Technology Specialist: Windows Sharepoint Services 3.0
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2008
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 3.5, ASP.NET Applications
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2005
- MCTS - Microsoft Certified Technology Specialist: Windows Vista
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 2.0
- MCDBA – Microsoft Certified Database Administrator (SQL Server 2000/OLAP/BI)
- MCAD – Microsoft Certified Application Developer .NET
- MCSA 2000 – Microsoft Certified System Administrator Windows 2000
- MCSA 2003 – Microsoft Certified System Administrator Windows 2003
- Microsoft Small and Medium Business Specialist
- MCP – Visual Basic e ASP
- MCT – Microsoft Certified Trainer
- SUN Java Trainer – Java Core Trainer Approved
- IBM Certified System Administrator – Lotus Domino 6.0/6.5

| | | |
|----------|--|-----------|
| 1 | Introdução | 4 |
| 1.1 | Conceito de Banco de Dados | 4 |
| 1.2 | Comandos SQL | 5 |
| 1.3 | Tipos de Dados | 5 |
| 1.3.1 | Regras de Nomenclatura e Escrita | 5 |
| 1.3.2 | Tipos Comuns | 6 |
| 1.3.3 | Tipos Definidos Pelo Usuário | 6 |
| 1.3.4 | Variáveis | 7 |
| 1.4 | Tipos de Objetos | 7 |
| 1.5 | Batch e Script | 7 |
| 2 | Consultas Básicas | 9 |
| 2.1 | SELECT...FROM | 9 |
| 2.1.1 | SELECT DISTINCT | 9 |
| 2.1.2 | CASE...WHEN...THEN...END | 10 |
| 2.2 | WHERE | 10 |
| 2.2.1 | Filtragem de Strings | 11 |
| 2.2.2 | Operador IN | 11 |
| 2.2.3 | Operadores e Precedência | 11 |
| 2.3 | ALIAS | 12 |
| 2.4 | ORDER BY | 12 |
| 2.5 | TOP...WITH TIES | 12 |
| 3 | Agrupando Dados | 14 |
| 3.1 | Funções de Agregação | 14 |
| 3.2 | GROUP BY | 14 |
| 3.3 | HAVING | 15 |
| 4 | Consultas com Múltiplas Tabelas | 16 |
| 4.1 | INNER JOIN | 16 |
| 4.2 | LEFT/RIGHT JOIN | 17 |
| 4.3 | FULL JOIN | 17 |
| 4.4 | JOIN com Chaves Composta | 18 |
| 4.5 | SELECT...INTO e Tabela Temporária | 18 |
| 5 | Subqueries | 20 |
| 5.1 | Geração de Coluna | 20 |
| 5.2 | Construção de WHERE | 21 |
| 5.3 | Construção de IN | 21 |
| 6 | Transações | 22 |

| | | |
|------|---------------------------|----|
| 6.1 | BEGIN...COMMIT...ROLLBACK | 22 |
| 6.2 | LOCKS | 22 |
| 7 | Inserindo Dados | 23 |
| 7.1 | INSERT com SELECT | 23 |
| 8 | Deletando Dados | 24 |
| 9 | Alterando Dados | 25 |
| 10 | Índices | 26 |
| 10.1 | Estrutura de Índices | 26 |
| 10.2 | Otimizando Performance | 26 |
| 11 | Diagramas | 27 |
| 11.1 | PUBS | 27 |
| 11.2 | Northwind | 28 |

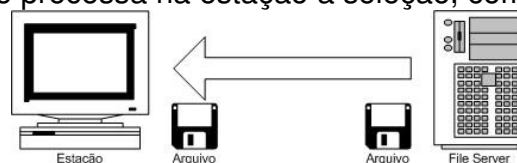
1 Introdução

1.1 Conceito de Banco de Dados

Antes dos bancos de dados eram utilizados arquivos em formato texto (*flat file*) ou binário proprietário, como por exemplo, o Clipper com dbf, Cobol com dat, etc.

Os arquivos de dados eram copiados em um servidor de arquivos e disponibilizados aos usuários por acesso em rede, ou seja, era dado o acesso ao diretório onde os arquivos estavam, impossibilitando controlar o que era feito com os dados, apenas podíamos controlar quem acessava, mas não as informações ou operações que efetuava.

Outra característica é que os dados eram processados localmente, se o usuário precisasse de uma lista com todos os clientes de São Paulo, o servidor de arquivos era acessado e copiava para a memória da estação o arquivo e processa na estação a seleção, como o esquema abaixo:

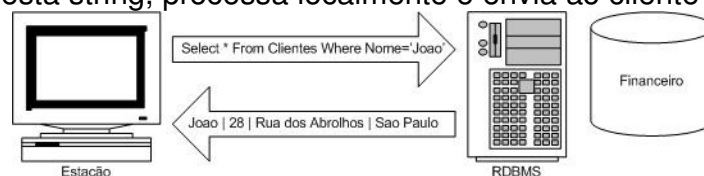


Enquanto os bancos de dados eram pequenos e monousuários este modelo funcionou bem, mas com a população de sistemas e clientes crescendo este modelo gerou lentidão e problemas de concorrência.

Para isso surgiram os primeiros servidores de banco de dados, arquitetura chamada de *client-server*, onde os dados residem em um servidor.

O cliente envia ao servidor uma string com a lista de dados, o nome da tabela e a condição.

O servidor recebe esta string, processa localmente e envia ao cliente apenas o resultado.



Como este processo trafega um pequeno comando e retorna dados já selecionados, o tráfego de rede melhorou muito. Também foi a solução para o problema da concorrência, uma vez que como um único servidor está com a posse dos dados, ele pode controlar quem acessa ou não a informação e não permitir alterações simultâneas em um mesmo dado, por controlar a fila de solicitações.

Após o surgimento dos servidores foi criado um padrão chamado de relacional (RDBMS), que envolve separar os dados em diferentes tabelas e juntá-los no momento da pesquisa. Também existe um outro padrão chamado de banco de dados orientados a objetos, mas este não é popular e possui um conceito diferente dos relacionais.

No início existiram diversos produtos para servidor como o ISAM, VMS, ADABAS e outros, sendo que cada um tinha o seu próprio modelo de solicitações e instruções. A fim de criar um padrão a IBM na década de 70 criou um padrão utilizado hoje em todos os bancos de dados que é o SQL, Structured Query Language ou Linguagem Estruturada para Consultas.

Atualmente o SQL está na padronização ANSI-SQL92, ou seja, quem regula o padrão dos comandos é o organismo da ISO e estamos no modelo gerado em 1992.

Todos os bancos de dados relacionais (RDBMS) utilizam esta linguagem, mas possui uma ou outra sintaxe diferente, como por exemplo, trazer cinco registros de uma tabela:

| | |
|-----------------------------|--|
| Microsoft SQL Server | SELECT TOP 5 * FROM CLIENTES |
| IBM DB2 UDB | SELECT * FROM CLIENTES FOR FIRST 5 ROWS ONLY |
| Oracle | SELECT * FROM CLIENTES WHERE @ROWCOUNT < 5 |

Esta apostila está voltada ao Microsoft SQL Server, sendo que algumas sintaxes poderão não ser as mesmas em outros bancos de dados.

Um único servidor RDBMS pode conter diversos databases, ou seja, bases de dados. Estes são repositórios para tabelas, objetos e dados, sendo que cada database é independente de outro.

Nesta apostila estaremos utilizando os databases Pubs (diagrama 10.1) e Northwind (diagrama 10.2) em todos os exemplos e exercícios propostos.

1.2 Comandos SQL

Os comandos SQL são separados em três famílias:

1. DDL – Data Definition Language
Utilizada para criar, deletar e alterar objetos como views, databases, stored procedures, etc.
2. DCL – Data Control Language
Permite controlar a segurança de dados definindo quem pode acessar cada operação em cada objeto.
3. DML – Data Manipulation Language
Comandos de manipulação que serão abrangidos nesta apostila.

| DDL | DCL | DML |
|-------------------------|-------------------------|--------------------------------------|
| CREATE ALTER DROP | GRANT REVOKE DENY | SELECT INSERT UPDATE DELETE |

DICA: Todos os comandos DDL e DCL seguem uma sintaxe básica, já os comandos DML são diferentes entre o SELECT e os outros três.

A sintaxe básica de um comando DDL é:

```
CREATE <tipo> <nome><definição>
CREATE TABLE Clientes (Nome CHAR(30), Telefone CHAR(8))
```

A sintaxe básica de um comando DCL é:

```
GRANT <operação> ON <nome do objeto> TO <usuário>
GRANT UPDATE ON Clientes TO João
```

1.3 Tipos de Dados

1.3.1 Regras de Nomenclatura e Escrita

Ao trabalhar com SQL é comum utilizarmos PascalCasing nos casos de variáveis, nomes de objetos, colunas e outras definições. Já no caso de palavras chaves segue-se o padrão maiúsculo em todo o comando. Veja o exemplo de como esta regra é aplicada:

```
SELECT Nome, Sobrenome, Idade
FROM Clientes
WHERE Cidade = 'Guarulhos'
```

Note a indentação utilizada para os comandos e também que em SQL strings são delimitadas por apóstrofo, ou aspas simples, e não por aspas comuns.

Comandos SQL não utilizam como delimitador o ENTER e sim o espaço, ou seja, não existe marca de fim de linha, o processador de consultas separa cada palavra e consulta a tabela de comandos, o que for diferente da tabela de comandos é considerado parâmetro ou objeto.

Ou seja, objetos que utilizem espaço em branco ou tenham o mesmo nome que uma instrução precisam ser delimitados por colchetes. Veja o exemplo abaixo de uma tabela com espaços e com palavra reservada:

```
/*
  Faz consulta na tabela de clientes
  Trazendo os dados mais importantes dos USA
*/
SELECT Nome, Sobrenome, [Order]      --Dados principais
FROM [Clientes USA]                  --Tabela de clientes
WHERE [State] = 'USA'                 --País desejado
```

Foi necessário utilizar os colchetes na palavra *order* e *state* por também serem palavras reservadas, bem como no nome da tabela por conter espaços.

DICA: Evite utilizar palavras reservadas e espaços em nomes de objetos.

Comentários podem ser feitos em bloco ou em linha. Comentários em linha são definidos por “--” e a linha de comando precisa continuar após, como o exemplo anterior. Comentários em bloco são delimitados por “/*” e finalizados em “*/”.

1.3.2 Tipos Comuns

Os tipos de dados entre os diferentes RDBMS são padronizados pelo ANSI-92, mas alguns bancos podem utilizar nomenclaturas diferentes. Segue a tabela de tipos comuns:

| Tipo Comum | MS-SQL | Definição |
|--|----------------------|---|
| binary varying | varbinary | Até 8.000 bytes, ocupação variável |
| character(<i>n</i>) | char(<i>n</i>) | Até 8.000 caracteres, ocupação fixa |
| character varying(<i>n</i>) | varchar(<i>n</i>) | Até 8.000 caracteres, ocupação variável |
| dec | decimal | -10 ³⁸ a 10 ³⁸ , tamanho fixo com casa decimal fixa |
| float[(<i>n</i>)] de <i>n</i> = 1-7 | real | -3.40E + 308 a 3.40E +308 |
| float[(<i>n</i>)] de <i>n</i> = 8-15 | float | -1.79E + 308 a 1.79E +308 |
| integer | int | -2.147.483.648 a 2.147.483.648 |
| blob | image | Até 2 GB de dados binários |
| clob | text | Até 2 GB de caracteres |
| national character(<i>n</i>) | nchar(<i>n</i>) | Até 4.000 caracteres, ocupação fixa (ocupa 2 bytes por caractere) |
| national character varying(<i>n</i>) | nvarchar(<i>n</i>) | Até 4.000 caracteres, ocupação variável (ocupa 2 bytes por caractere) |
| national text | ntext | Até 1 GB de caracteres (ocupa 2 bytes por caractere) |
| numeric | decimal | -10 ³⁸ a 10 ³⁸ , tamanho fixo com casa decimal fixa |

Note que alguns tipos levam a letra “n” indicando que são unicode, ou seja, para permitir uma tabela de caracteres maior que 256 estes tipos utilizam 2 bytes para cada letra ao invés de 1 byte convencional do ASCII.

Tipos variáveis são melhores para textos pois ocupam apenas o espaço digitado, enquanto os tipos fixos ocupam espaço em branco quando não digitado.

Alguns tipos de dados tem o tamanho definido na criação, como por exemplo *char*, *varchar* e *nvarchar*.

1.3.3 Tipos Definidos Pelo Usuário

Alem dos tipos principais temos também os tipos definidos pelo usuário que são baseados nos tipos do sistema. Por exemplo, o tipo CEP é uma variação de 9 caracteres, levando-se em conta o traço, como o exemplo abaixo:

```
sp_addtype 'CEP', 'char(9)'
```

É importante definir no início da criação os tipos a serem usados e os que podem ser criados para não ter variações em um mesmo dado, como por exemplo o CEP pode ser utilizado como 8 dígitos ou 8 caracteres. Criar o tipo CEP limita possíveis confusões.

1.3.4 Variáveis

Variáveis podem ser definidas como locais ou globais.

Variáveis locais são aquelas que só podem ser vistas pelo usuário que a criou e as globais podem ser ligadas por qualquer usuário conectado ao banco de dados.

As variáveis são criadas utilizando uma arroba (@) para locais e duas arrobas para globais (@@), como os exemplos a seguir:

```
DECLARE @Nome CHAR(30)
DECLARE @@Dolar DECIMAL(10,3)
```

A palavra chave *Declare* é utilizada para criar os dois tipos de variáveis, sendo seguido do nome e do tipo.

Para atribuir valor a uma variável utilizamos a instrução *Set*, como o exemplo:

```
SET @Nome = 'Joao'
SET @@Dolar = 3.04
PRINT @Nome
PRINT @@Dolar
```

DICA: O MSSQL 2000 não possui mais a definição para variáveis globais.

1.4 Tipos de Objetos

Objetos no SQL são criados com os comandos DDL e destacam-se:

- **View**
Uma consulta pré-gravada que facilita a reutilização de consultas complexas
- **Stored Procedure (SP)**
São programas feitos em T-SQL para executar tarefas diversas, com estrutura similar ao Pascal e instruções como *IF*, *Else*, *Begin*, *End*, etc.
- **User Defined Function (UDF)**
São SPs que recebem parâmetros e retornam um parâmetro ou uma tabela
- **Triggers**
São SPs executadas automaticamente quando uma ação é efetuada sobre uma tabela, como por exemplo, executar todas as vezes que deletar um cliente
- **Constraints**
Regras de validação, como por exemplo preenchimento obrigatório, valor não duplicado, delimitador de valor, etc.

1.5 Batch e Script

Batch são seqüências de comandos executadas em um único bloco.

Não é necessário enviar os comandos separadamente, eles podem ser enviados ao servidor e este se encarrega de dividir os diferentes comandos e executar de forma assíncrona.

Batch melhora a performance por não obrigar o cliente a enviar comandos um a um.

Entre alguns comandos, principalmente os DDL utilizamos a instrução *go* entre cada um, a fim de fazer o processo tornar-se síncrono, ou seja, uma instrução terminar antes de outra iniciar. Veja o exemplo abaixo:

```
CREATE TABLE Clientes
```

```
(Nome char(30),  
  Cidade varchar(50),  
  Estado char(2))  
  
GO  
INSERT CLIENTES VALUES('Joao', 'Guarulhos', 'SP')  
SELECT *  
FROM CLIENTES
```

Note que a segunda instrução é a inserção do registro na tabela, portanto o uso do *go* força a tabela de clientes a ser criada para depois executar os comandos seguintes. Se não houvesse o *go* correríamos o risco do *insert* ser executado junto com o *create*, e neste caso a tabela ainda não existiria.

Scripts são arquivos textos que contêm um batch para ser executado.

Diversas ferramentas podem executar o script, por ler o seu conteúdo e enviar para o servidor a fim de execução.

Também é possível executar comandos em string como o exemplo abaixo, utilizando a instrução *exec*:

```
--COMANDOS DINAMICOS  
DECLARE @COMANDO VARCHAR(2000)  
DECLARE @TABELA VARCHAR(100)  
DECLARE @DATABASE VARCHAR(100)  
SET @TABELA='CUSTOMERS'  
SET @DATABASE='NORTHWIND'  
SET @COMANDO='USE ' + @DATABASE +  
              ' SELECT * FROM ' + @TABELA  
EXECUTE(@COMANDO)  
PRINT @COMANDO
```


2 Consultas Básicas

2.1 SELECT...FROM

A cláusula *select* é obrigatoriamente seguida de uma cláusula *from*. Enquanto a primeira cláusula define quais os dados desejados, a segunda indica de onde serão obtidos.

A fonte de dados pode ser uma tabela, uma view ou uma UDF.

Como exemplo de um comando para seleção de dados, veja a instrução e o resultado:

```
SELECT au_id, au_lname, au_fname, phone, city, state
FROM Authors
```

| au_id | au_lname | au_fname | phone | city | state |
|-------------|----------|----------|--------------|------------|-------|
| 172-32-1176 | White | Johnson | 408 496-7223 | Menlo Park | CA |
| 213-46-8915 | Green | Marjorie | 415 986-7020 | Oakland | CA |
| 238-95-7766 | Carson | Cheryl | 415 548-7723 | Berkeley | CA |
| 267-41-2394 | O'Leary | Michael | 408 286-2428 | San Jose | CA |
| 274-80-9391 | Straight | Dean | 415 834-2919 | Oakland | CA |
| 341-22-1782 | Smith | Meander | 913 843-0462 | Lawrence | KS |
| ... | | | | | |

(23 row(s) affected)

O *select* acima não ordenou os dados nem alterou qualquer característica original, mas trouxe apenas as colunas que foram definidas na lista.

Na tabela *authors* da *pubs* (diagrama 10.1) existem outras colunas que não apenas as seis utilizadas nesta consulta, bem como a ordem das colunas na tabela não afetam em nada a ordem em que um *select* solicita o retorno destes dados.

Neste caso utilizamos o database *pubs*, e para isso precisamos definir o database que será utilizado. Podemos definir o database de duas formas, a primeira é utilizando o caminho completo, como a seguir:

```
SELECT au_id, au_lname, au_fname, phone, city, state
FROM Server.Pubs.dbo.Authors
```

Neste caso definimos o nome do servidor, o database desejado e o esquema ou usuário que criou a tabela. Como padrão o esquema é sempre *dbo* na maior parte dos RDBMS.

A segunda forma de definir o banco de dados é utilizando o comando *Use Pubs*. Não é necessário utilizar este comando em todos os *select* como no exemplo anterior, uma vez que o *use* é um comando de sessão, ou seja, ao abrir a conexão com o servidor enviamos o *use* e não mais precisamos definir o database no *from*.

2.1.1 SELECT DISTINCT

Para evitar repetições nos dados retornados, quando estes possuem duplicações podemos utilizar o *distinct* que tem como função selecionar apenas as linhas em que todas as colunas não coincidam entre si. Veja o exemplo abaixo:

```
SELECT DISTINCT city, state
FROM Authors
```

No exemplo acima serão retornados apenas os registros em que a cidade mais o estado juntos sejam diferentes, evitando uma repetição na lista de cidades e estados onde existem autores.

2.1.2 CASE...WHEN...THEN...END

Utilizamos esta estrutura para dados condicionais, como por exemplo, caso o valor seja 10 mostrar uma mensagem e se for diferente de 10 outra mensagem.
No exemplo abaixo utilizamos o *case* para mostrar nomes em português para os nomes originais em inglês.

```
SELECT TITLE_ID,
       CASE TYPE
         WHEN 'BUSINESS' THEN 'NEGOCIOS'
         WHEN 'POPULAR_COMP' THEN 'INFORMATICA'
         WHEN 'PSYCHOLOGY' THEN 'PSICOLOGIA'
         ELSE 'OUTROS'
       END AS TIPO
FROM TITLES
```

```
TITLE_id TIPO
-----
BU1032 NEGOCIOS
BU1111 NEGOCIOS
BU2075 NEGOCIOS
BU7832 NEGOCIOS
MC2222 OUTROS
MC3021 OUTROS
MC3026 OUTROS
PC1035 INFORMATICA
```

Note que não existe virgula entre os operadores *when* pois eles não são novas colunas e sim continuação da condição. Na ultima utilização do *case* com o operador *else* criamos o padrão caso o tipo do livro não se encaixe nas três condições anteriores.

2.2 WHERE

Esta clausula indica as condições da consulta.

As condições melhoram a performance por permitir que o *engine* do banco de dados utilize os índices para organizar a forma de procura, não fazendo método bolha e sim de ponteiro.

Abordaremos índices em detalhes no módulo 9.

Segue a lista operadores, função e exemplo:

| Operador | Função | Exemplo |
|----------|---|--------------------------------|
| = | Altera valor de variáveis CTS e compara igualdade | Idade = 30 |
| <> | Idade diferente de 30 | Idade <> 30 |
| > e >= | Maior (31 para frente) e maior ou igual (inclui o 30) | Idade > 30 Idade >= 30 |
| < e <= | Menor (29 para baixo) e menor ou igual (inclui o 30) | Idade < 30 Idade <= 30 |
| Between | Idade que esteja entre 50 e 90 inclusive | Idade Between 50 And 90 |
| Not | Negação de uma condição | Idade Not Between 50 And 90 |
| Is | Compara igualdade entre tipos | Nome is Null |
| And | Significa que as comparações devem ser todas verdadeiras | Idade = 30 And Nome = "Fulano" |
| Or | Significa apenas uma das comparações precisa ser verdadeira | Idade = 30 Or Nome = "Fulano" |

Alguns exemplos de *where* podem ser vistos abaixo:

```
--UTILIZAÇÃO DE NUMERICOS
Use Northwind
SELECT *
FROM PRODUCTS
WHERE UNITPRICE BETWEEN 10 AND 12
SELECT *
FROM PRODUCTS
WHERE UNITPRICE NOT BETWEEN 10 AND 12
SELECT *
```

```
FROM PRODUCTS
WHERE UNITPRICE = 10 Or UNIPRICE = 12
```

2.2.1 Filtragem de Strings

Para filtrar strings temos uma situação e instrução própria, uma vez que números tem um valor absoluto e textos não. Por exemplo, eu procuro pela idade de 31 anos exatamente, mas nem sempre sei o nome exato de uma pessoa.

A instrução *like* tem como diferencial o uso dos coringas “%” para qualquer coisa na substituição (similar ao “*” do DOS), “[” e “]” para definir range. Os exemplos abaixo definem bem estas regras:

```
USE NORTHWIND
SELECT *
FROM CUSTOMERS
WHERE CustomerID='ANTON' --Nome Anton é o unico que será encontrado
SELECT *
FROM CUSTOMERS
WHERE COMPANYNAME LIKE '[B-E]%' --Iniciem entre B e E
SELECT *
FROM CUSTOMERS
WHERE COMPANYNAME LIKE '[BMF]%' --Iniciem em B, M ou F
SELECT *
FROM CUSTOMERS
WHERE CONTACTTITLE LIKE '%ETI%' --Contenham ETI em qualquer lugar
SELECT *
FROM CUSTOMERS
WHERE CONTACTTITLE LIKE '%ETI%ASS%' --Contenham ETI e ASS
SELECT *
FROM CUSTOMERS
WHERE CONTACTNAME LIKE 'L[A]%' --Iniciem em L e a segunda letra A ou I
```

2.2.2 Operador IN

Este operador é uma alternativa a uma extensa lista de *or*, uma vez que sua utilização é definir uma lista dos valores numéricos ou strings desejados.

Veja abaixo a comparação entre um comando com *or* e com *in*:

```
SELECT *
FROM CUSTOMERS
WHERE COUNTRY IN ('BRAZIL','ARGENTINA')
SELECT *
FROM CUSTOMERS
WHERE COUNTRY='BRAZIL' OR COUNTRY='ARGENTINA'
```

A diferença entre os dois operadores fica mais evidente se fossem dez países, por exemplo, onde o comando com *or* seria muito maior e de entendimento mais complicado do que o comando com *in*.

2.2.3 Operadores e Precedência

Assim como em cálculos matemáticos, existe uma precedência em processamento de variáveis e valores. Por exemplo, na equação $1+2*3/4$ o resultado será dois e meio uma vez que multiplicação e divisão são executadas antes da adição e subtração.

Para servir de referencia siga a seguinte regra: *, /, %, +, -, And e Or.

Como alternativa para controle de precedência utilize parênteses. Por exemplo ao invés de escrever *A Or B And C* escreva *(A Or B) And C*. Vamos entender a diferença.

No primeiro exemplo B e C tem que ser igual e A não faz diferença. No segundo exemplo A e B não fazem diferença, qualquer um deles pode ser combinado com C. Aqui fica clara a diferença feita pelo controle de precedência.

2.3 ALIAS

Alias são literalmente apelidos, pois algumas colunas em um *select* podem ser a soma de outras colunas, resultando em uma nova coluna. Esta nova coluna por padrão recebe o nome *Expr_000x* e podemos renomear com a instrução *as*.

```
SELECT COMPANYNAME, REGION+'/' + COUNTRY AS LOCAL
FROM CUSTOMERS
SELECT COMPANYNAME, LOCAL=REGION+'/' + COUNTRY
FROM CUSTOMERS
```

Nos dois exemplos acima região foi concatenado ao país para formar uma nova coluna, que chamamos de local. Note que tanto podemos usar o *as* quanto colocar o nome desejado e o sinal de igual com o conteúdo.

Alias também pode ser utilizado em tabelas, onde após o nome da tabela colocamos o *as* seguido do nome referencial. É muito utilizado nos *joins* tratados no módulo 4.

2.4 ORDER BY

Com esta cláusula conseguimos definir em que ordem queremos o retorno dos dados.

No MSSQL não é necessário que a coluna esteja na lista do *select* para ser ordenada, enquanto em outros bancos de dados a coluna utilizada para ordenação tem que constar na lista de colunas.

Veja alguns exemplo abaixo:

```
SELECT COMPANYNAME, CONTACTNAME
FROM CUSTOMERS
ORDER BY COMPANYNAME
SELECT COMPANYNAME AS EMPRESA, CONTACTNAME
FROM CUSTOMERS
ORDER BY 1 DESC
```

Os dois exemplos ordenam pela mesma coluna, mas a diferença é que no *order* pode ser utilizado tanto o nome da coluna quanto a posição dela na lista do *select*.

Utilizar o número da coluna é interessante quando temos colunas calculadas, uma vez que a coluna só existirá após o processamento e o MSSQL faz a ordenação antes de calcular novas colunas por padrão.

2.5 TOP...WITH TIES

Uma das funções que facilitam muito o trabalho com tabelas ordenadas é trazer um número limitado de registros em uma tabela.

Por exemplo, imagine que precisamos saber os 10 produtos mais vendidos. Processamos um *select* que ordena a tabela por quantidade e com o *top* especificamos quantos registros desejamos no resultado final, como o exemplo abaixo:

```
SELECT DISTINCT TOP 5 PRODUCTID, QUANTITY
FROM [ORDER DETAILS]
ORDER BY QUANTITY DESC
```

Algumas vezes podemos ter a situação em que o primeiro até o quarto item tiveram a mesma venda e do quinto ao sétimo a segunda maior venda. Neste caso o *top 5* irá retornar os 5 maiores produtos

e na quinta linha um dos três que venderam em segundo. Esta comparação pode ser prejudicial quando analisamos comissão, por exemplo.

Para resolver casos em que a ultima linha contenha repetições, ou empates, e estes não sejam omitidos acrescentamos o *with ties*:

```
SELECT DISTINCT TOP 5 WITH TIES PRODUCTID, QUANTITY  
FROM [ORDER DETAILS]  
ORDER BY QUANTITY DESC
```

Agora não retornaram mais apenas cinco produtos, mas sim sete, uma vez que o ultimo produto possuía outros dois com a mesma colocação.

A base para definir os empates no *with ties* é a coluna utilizada para fazer o *order*.

3 Agrupando Dados

Os operadores são utilizados para geração de valores agrupados.

Estas funções listadas abaixo são utilizadas para totalizar, somar, gerar relatórios, estatísticas e outras funções onde não interessa linha a linha, mas sim um resumo.

| Operador | Função |
|----------|---|
| AVG | Média aritmética |
| COUNT | Conta o numero de ocorrencias de linhas |
| MAX | Maior valor na coluna |
| MIN | Menor valor na coluna |
| SUM | Soma todos os valores da coluna |
| STDEV | Desvio padrão estatístico na coluna |
| STDEVP | Desvio padrão populacional na coluna |
| VAR | Variação estatística dos valores da coluna |
| VARP | Variação populacional dos valores da coluna |

Deve-se tomar cuidado com o resultado das funções de agregação, com exceção do *count*, pois elas desconsideram os valores nulos. Por exemplo, se houver 20 clientes, destes apenas 16 tem dados e 4 estão nulos. A média será a soma dos valores divididos por 16 e não por 20 como deveria ser, gerando uma média incorreta.

DICA: Para resolver o problema dos nulos usamos a função isnull(coluna,valor), onde se a coluna definida estiver nula, assume o outro valor. Portanto isnull(compras,0) evita o nulo por substitui-lo por um valor 0.

3.1 Funções de Agregação

Para utilizar as funções de agregação utilizamos a coluna a ser agregada dentro da função de agregação. Veja o exemplo abaixo:

```
SELECT MAX(UNITPRICE),MIN(UNITPRICE),
       AVG(QUANTITY),COUNT(*),COUNT(DISTINCT PRODUCTID),
       COUNT(DISTINCT ORDERID)
FROM [ORDER DETAILS]
```

```
-----
263.5000      2.0000      23      2155      77      830
```

Ao utilizar as funções de agregação retornam os dados agregados em uma única linha, e todos os dados precisam ter uma agregação para poderem ser utilizados com funções agregadas. O motivo desta sintaxe é que as colunas *unitprice* e *quantity*, por exemplo, estão retornando o menor valor e média respectivamente. Se for colocada a coluna código do produto sem qualquer agregação, o *engine* do banco de dados não teria como retornar um valor por não ter recurso para encontrar o apropriado.

3.2 GROUP BY

Nos casos em que colunas não agregadas serão utilizadas para criarem grupos de somas, utilizamos a instrução *group by*.

Esta instrução irá definir qual coluna da consulta será utilizada para fazer a quebra, ou subgrupo das agregações solicitadas.

```
SELECT PRODUCTID, MAX(UNITPRICE),MIN(UNITPRICE),
       AVG(QUANTITY),COUNT(*),COUNT(DISTINCT PRODUCTID),
```

```

COUNT(DISTINCT ORDERID)
FROM [ORDER DETAILS]
GROUP BY PRODUCTID

```

PRODUCTID

| | | | | | | |
|-----|---------|---------|----|----|---|----|
| 61 | 28.5000 | 22.8000 | 25 | 24 | 1 | 24 |
| 3 | 10.0000 | 8.0000 | 27 | 12 | 1 | 12 |
| 32 | 32.0000 | 25.6000 | 19 | 15 | 1 | 15 |
| 6 | 25.0000 | 20.0000 | 25 | 12 | 1 | 12 |
| 41 | 9.6500 | 7.7000 | 20 | 47 | 1 | 47 |
| ... | | | | | | |

Note que a coluna *productid* foi incluída na consulta sem nenhuma função de agregação, portanto ela foi utilizada no grupo, gerando os dados anteriores agora com o código e por produto.

DICA: As colunas em um Select deve estar com agregação ou no Group By quando alguma coluna do Select possuir agregação.

3.3 HAVING

Assim como podemos filtrar linhas de uma tabela com a instrução *where*, podemos fazer o mesmo quando os dados foram agrupados utilizando *having*.

A diferença básica entre o *where* e o *having* é que primeiro faz o filtro ao selecionar os registros para serem somados, sobre as linhas originais.. O segundo faz a filtragem quando as agregações já foram efetuadas, portanto, sobre o valor agrupado.

Veja os dois exemplos abaixo:

```

SELECT PRODUCTID, SUM(QUANTITY), COUNT(*)
FROM [ORDER DETAILS]
WHERE QUANTITY > 100
GROUP BY PRODUCTID

```

```

SELECT PRODUCTID, SUM(QUANTITY), COUNT(*)
FROM [ORDER DETAILS]
GROUP BY PRODUCTID
HAVING SUM(QUANTITY) > 100

```

Note que o primeiro exemplo irá fazer a soma apenas das linhas em que as vendas tenham quantidade maior que 100 itens. Estas vendas que foram agregadas são as vendas individuais a cada pedido.

A segunda consulta não leva em conta o item individual do pedido e sim a soma total de cada produto. Neste caso, primeiro somamos todas as vendas e agrupamos por produtos, e no resultado teremos os produtos que a soma total foi maior do que 100 itens vendidos.

Podemos combinar as duas instruções para refinar dados, como o exemplo abaixo:

```

SELECT PRODUCTID, SUM(QUANTITY), COUNT(*)
FROM [ORDER DETAILS]
WHERE UNITPRICE > 15
GROUP BY PRODUCTID
HAVING SUM(QUANTITY) > 100

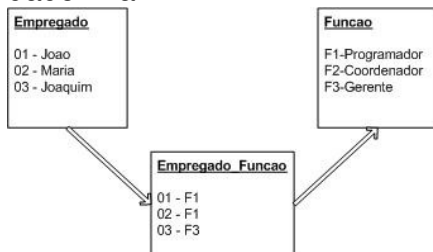
```

A diferença é que a consulta cima irá somar na tabela apenas os produtos com preço unitário maior do que 15, e após estes produtos já terem sido somados selecionamos do resultado apenas os que venderam acima de 100 itens. Resumindo o resultado teremos todos os produtos que o preço é maior do que 15 reais e venderam mais do que 100 itens.

4 Consultas com Múltiplas Tabelas

Todos os modelos vistos de consultas até o momento incluíam dados de apenas uma tabela. Seguindo os modelos de normalização de dados, separamos em diferentes tabelas quando os dados não são diretamente relacionados ou dependentes.

Veja no exemplo abaixo uma situação onde necessitamos juntar tabelas para podermos chegar no dado final:



```

SELECT e.NOME, f.FUNCAO
FROM EMPREGADO AS e
INNER JOIN EMPREGADO_FUNCAO AS ef
  ON e.CODIGO=ef.CODIGO
INNER JOIN FUNCAO AS f
  ON f.CODFUNC = e.CODFUNC
  
```

Note que para podermos ter o nome do funcionário e a função que ele desempenha foi necessário utilizar uma terceira tabela que contem apenas o código do funcionário e o código da sua função. O *engine* para fazer este processo pega o código do funcionário na tabela *empregado* e procura este funcionário na tabela *empregado_funcao* de onde retira o código da função e o utiliza para procurar na tabela *funcao* para poder encontrar o nome.

Este processo de juntar dados de uma tabela com a outra utilizando uma coluna como guia é chamado de *join*. Existem quatro tipos diferentes de *join* no *MSSQL*, mas abordaremos os três tipos principais e padronizados.

A sintaxe básica é colocar após a primeira tabela a instrução do tipo de *join* e na seqüência a instrução *on* que define qual é a coluna na tabela A que permite a junção da tabela B.

4.1 INNER JOIN

Este é o mais comum utilizado e sua característica é só trazer registros que contenham correspondência nas outras tabelas utilizadas.

O comando utilizado anteriormente também era um exemplo de *inner join*, uma vez que todos os funcionários possuem função, portanto todos eles retornariam. Mas se existisse um funcionário de código 04 e este não tivesse o registro *empregado_funcao* só retornariam três registros.

Veja o exemplo que contem uma estrutura similar no database *pubs*:

```

SELECT A.AU_FNAME, A.AU_LNAME, T.TITLE, T.TYPE
FROM TITLEAUTHOR AS TA
INNER JOIN AUTHORS AS A
  ON TA.AU_ID = A.AU_ID
INNER JOIN TITLES AS T
  ON TA.TITLE_ID = T.TITLE_ID
  
```

| AU_FNAME | AU_LNAME | TITLE | TYPE |
|----------|----------------|---|--------------|
| Abraham | Bennet | The Busy Executive's Database Guide | business |
| Reginald | Blotchet-Halls | Fifty Years in Buckingham Palace Kitchens | trad_cook |
| Cheryl | Carson | But Is It User Friendly? | popular_comp |
| Michel | DeFrance | The Gourmet Microwave | |
| ... | | | |

Deve-se lembrar que a utilização do *inner join* pode não trazer todos os registros da tabela original, principalmente se nas outras tabelas relacionadas não existir correspondente.

4.2 LEFT/RIGHT JOIN

As instruções *Left Join* e *Right Join* são utilizadas para resolver a não correspondência de dados entre diferentes tabelas.

Por exemplo, na tabela *authors* da *pubs* poderia não haver livros editados para algum autor, assim como posso ter um registro de livro na *titles* que ainda não tenha sido designado a nenhum dos autores. Veja o exemplo abaixo onde todos os autores irão retornar, mesmo os que não tenham livros editados:

```
SELECT A.AU_FNAME, A.AU_LNAME, T.TITLE, T.TYPE
FROM AUTHORS AS A
LEFT JOIN TITLEAUTHOR AS TA
  ON TA.AU_ID = A.AU_ID
LEFT JOIN TITLES AS T
  ON TA.TITLE_ID = T.TITLE_ID
```

| AU_FNAME | AU_LNAME | TITLE |
|-------------|----------|-------|
| Morningstar | Greene | NULL |
| Heather | McBadden | NULL |
| Meander | Smith | NULL |
| Dirk | Stringer | NULL |

Os quatro escritores acima ainda não escreveram qualquer livro pela editora, portanto só apareceram em virtude do *left join*. Podemos entender o *left* e o *right* como designando a tabela a esquerda ou direita como a principal, portanto não importando a outra tabela ter ou não sua correspondência.

Invertendo a situação, queremos ver os livros que não tenham sido designados a escritores. Neste caso a tabela de *titles* está a direita das outras, então basta alterar o *left* por *right* para vermos a diferença:

```
SELECT A.AU_FNAME, A.AU_LNAME, T.TITLE
FROM AUTHORS AS A
RIGHT JOIN TITLEAUTHOR AS TA
  ON TA.AU_ID = A.AU_ID
RIGHT JOIN TITLES AS T
  ON TA.TITLE_ID = T.TITLE_ID
```

| AU_FNAME | AU_LNAME | TITLE |
|----------|----------|-------------------------------------|
| NULL | NULL | The Psychology of Computer Cooking |
| Abraham | Bennet | The Busy Executive's Database Guide |

Note agora o livro da primeira linha que antes não estava relacionado por não ter o escritor, mas agora como a tabela de livros foi definida como a principal por esta a direita (*right*) este livro foi relacionado.

4.3 FULL JOIN

Ainda restou nos exemplos acima uma situação não resolvida.

Quanto utilizamos o *left join* conseguimos saber os autores que não estavam com nenhum livro editado e com o *right join* conseguimos os livros sem autores. Mas agora precisamos saber em uma única relação tanto os livros sem autores quanto os autores sem livros.

O exemplo abaixo utilizando o *full join* designa as duas tabelas relacionadas como sendo principais e retornando dados mesmo sem o correspondente:

```
SELECT A.AU_FNAME, A.AU_LNAME, T.TITLE
FROM AUTHORS AS A
FULL JOIN TITLEAUTHOR AS TA
  ON TA.AU_ID = A.AU_ID
FULL JOIN TITLES AS T
  ON TA.TITLE_ID = T.TITLE_ID
```

| AU_FNAME | AU_LNAME | TITLE |
|----------|--------------|-------------------------------------|
| NULL | NULL | The Psychology of Computer Cooking |
| Abraham | Bennet | The Busy Executive's Database Guide |
| Akiko | Yokomoto | Sushi, Anyone? |
| Albert | Ringer | Is Anger the Enemy? |
| Albert | Ringer | Life Without Fear |
| Ann | Dull | Secrets of Silicon Valley |
| Anne | Ringer | Is Anger the Enemy? |
| Anne | Ringer | The Gourmet Microwave |
| Burt | Gringlesby | Sushi, Anyone? |
| Charlene | Locksley | Emotional Security: A New Algorithm |
| Charlene | Locksley | Net Etiquette |
| Cheryl | Carson | But Is It User Friendly? |
| Dean | Straight | Straight Talk About Computers |
| Dirk | Stringer | NULL |
| Heather | McBadden | NULL |
| Innes | del Castillo | Silicon Valley Gastronomic Treats |

Note na relação que temos os livros sem autores e os autores sem livros, resolvendo problema das linhas sem correspondências.

4.4 JOIN com Chaves Composta

Em alguns casos a chave para o relacionamento entre tabelas não é uma única coluna mas formada por múltiplas.

Neste caso podemos utiliza o *join* com o operador *and* para fazer a junção composta.

Também pode ser utilizado o *and* para fazer filtro, apesar de no exemplo abaixo também poderia ter sido utilizado o *where*:

```
SELECT A.AU_FNAME, A.AU_LNAME, T.TITLE, T.TYPE
FROM TITLEAUTHOR AS TA
INNER JOIN AUTHORS AS A
  ON TA.AU_ID = A.AU_ID
INNER JOIN TITLES AS T
  ON TA.TITLE_ID = T.TITLE_ID AND T.TYPE='BUSINESS'
```

4.5 SELECT...INTO e Tabela Temporária

O resultado de um *select* pode ser transformado em uma tabela utilizando a instrução *into*, conforme o exemplo abaixo:

```
SELECT A.AU_FNAME, A.AU_LNAME, T.TITLE, T.TYPE
INTO NOVATABELA
FROM TITLEAUTHOR AS TA
INNER JOIN AUTHORS AS A
  ON TA.AU_ID = A.AU_ID
INNER JOIN TITLES AS T
  ON TA.TITLE_ID = T.TITLE_ID
```

A tabela *novatabela* foi gerada com o resultado do *select* executado. Este modelo demonstrado tem a limitação para uso de ser uma tabela fixa.

Para criar uma tabela sem que esta esteja fisicamente no database utilizamos tabelas temporárias. Estas ficam alocadas durante a utilização em um database chamado *tempdb*.

Segue a sintaxe do comando *select* criando uma tabela temporária:

```
SELECT A.AU_FNAME, A.AU_LNAME, T.TITLE, T.TYPE  
INTO #NOVATABELA  
FROM TITLEAUTHOR AS TA  
INNER JOIN AUTHORS AS A  
ON TA.AU_ID = A.AU_ID  
INNER JOIN TITLES AS T  
ON TA.TITLE_ID = T.TITLE_ID
```

Note que utilizando um sinal “#” criamos uma tabela temporária de sessão, ou seja, enquanto o usuário estiver conectado, sendo automaticamente destruída ao ser encerrada a conexão. Com dois sinais “##” criamos a tabela pública, todos os usuários conectados a enxergam e esta é destruída quando não houver nenhuma conexão ativa no servidor.

5 Subqueries

Em algumas situações precisamos ter o dado de uma tabela para podermos criar uma consulta, utilizando os dados desta outra tabela em uma coluna ou nas condições. A este processo chamamos de *subquery*.

Subqueries podem ser utilizadas na geração de colunas, na geração de tabela fonte, na geração de *in* e cláusulas *where* em geral. Abordaremos os usos mais comuns das *subqueries*.

5.1 Geração de Coluna

Exemplificando a utilização de uma *subquery* para soma dos itens vendidos mês a mês, gerando um relatório de vendas, veja o exemplo abaixo:

```
SELECT C.COMPANYNAME,  
  (SELECT SUM(QUANTITY*UNITPRICE)  
   FROM ORDERS AS O  
   INNER JOIN [ORDER DETAILS] AS OD  
     ON O.ORDERID = OD.ORDERID  
   WHERE O.CUSTOMERID = C.CUSTOMERID  
     AND MONTH(ORDERDATE)=1  
     AND YEAR(ORDERDATE)=1998) AS JANEIRO,  
  (SELECT SUM(QUANTITY*UNITPRICE)  
   FROM ORDERS AS O  
   INNER JOIN [ORDER DETAILS] AS OD  
     ON O.ORDERID = OD.ORDERID  
   WHERE O.CUSTOMERID = C.CUSTOMERID  
     AND MONTH(ORDERDATE)=2  
     AND YEAR(ORDERDATE)=1998) AS FEVEREIRO,  
  (SELECT SUM(QUANTITY*UNITPRICE)  
   FROM ORDERS AS O  
   INNER JOIN [ORDER DETAILS] AS OD  
     ON O.ORDERID = OD.ORDERID  
   WHERE O.CUSTOMERID = C.CUSTOMERID  
     AND MONTH(ORDERDATE)=3  
     AND YEAR(ORDERDATE)=1998) AS MARCO,  
  (SELECT SUM(QUANTITY*UNITPRICE)  
   FROM ORDERS AS O  
   INNER JOIN [ORDER DETAILS] AS OD  
     ON O.ORDERID = OD.ORDERID  
   WHERE O.CUSTOMERID = C.CUSTOMERID  
     AND MONTH(ORDERDATE)=4  
     AND YEAR(ORDERDATE)=1998) AS ABRIL,  
  (SELECT SUM(QUANTITY*UNITPRICE)  
   FROM ORDERS AS O  
   INNER JOIN [ORDER DETAILS] AS OD  
     ON O.ORDERID = OD.ORDERID  
   WHERE O.CUSTOMERID = C.CUSTOMERID  
     AND MONTH(ORDERDATE)=5  
     AND YEAR(ORDERDATE)=1998) AS MAIO,  
  (SELECT SUM(QUANTITY*UNITPRICE)  
   FROM ORDERS AS O  
   INNER JOIN [ORDER DETAILS] AS OD  
     ON O.ORDERID = OD.ORDERID  
   WHERE O.CUSTOMERID = C.CUSTOMERID  
     AND MONTH(ORDERDATE)=6  
     AND YEAR(ORDERDATE)=1998) AS JUNHO  
FROM CUSTOMERS AS C  
ORDER BY COUNTRY
```

| COMPANYNAME JUNHO | JANEIRO | FEVEREIRO | MARCO | ABRIL | MAIO | |
|---|------------------|---------------------|-------------------|-----------------|--------------|--------------|
| Cactus Comidas para llevar NULL | 477.0000 | 150.0000 | 644.8000 | 305.0000 | NULL | |
| Océano Atlántico Ltda. Rancho grande | NULL 932.0000 | 30.0000 686.7000 | 3001.0000 NULL | NULL 76.0000 | NULL NULL | NULL NULL |
| Ernst Handel NULL | 8195.5000 | 6379.4000 | 6221.5000 | 16584.5000 | 5218.0000 | |

Como pode ser notado no *select*, cada coluna referente a um mes na verdade é um *select* realizado na tabela de pedidos, onde a condição é a coluna da tabela *customers*.

Uma importante ressalva a este modo de uso é que a cada cliente foram executados os seis *selects* internos. Ou seja, se a tabela contiver 1000 registros serão executados 6000 *selects* na tabela de pedidos, o que pode gerar um *overhead* muito grande no servidor.

5.2 Construção de WHERE

As utilizações mais práticas de *subqueries* é para criação de condições.

Estas duas utilizações são processadas antes do *select* principal ser executado, já que ele gera um único valor ou retorno. Portanto, a utilização de *subquerie* no *in* e no *where* são rápidas e não ocasionam *overhead*.

O exemplo abaixo demonstra a utilização da *subquerie* para geração da condição:

```
SELECT *
FROM ORDERS
WHERE ORDERDATE = (SELECT MAX(ORDERDATE)
FROM ORDERS)
```

Neste exemplo trouxemos da tabela *orders* a maior data de pedido existente e utilizamos esta data para listar os pedidos desta data. Este exemplo é útil quando precisamos listar todos os pedidos realizados no ultimo movimento registrado.

5.3 Construção de IN

Como já abordado nos módulos anteriores, o *in* gera uma lista de valores aceitos, substituindo de forma inteligente uma cadeia de operadores *or*.

O exemplo abaixo utiliza a *subquerie* para gerar uma lista das cidades que existem no Brasil e utiliza esta lista para trazer a relação de clientes que estão nestas cidades.

```
SELECT *
FROM CUSTOMERS
WHERE CITY
IN (SELECT CITY
FROM CUSTOMERS
WHERE COUNTRY='BRAZIL')
```

Um exemplo mais interessante para esta situação é trazer o nome de todos os clientes que tiveram pedidos com todos os produtos somados maior do que 5000 reais. Neste caso utilizamos uma *subquerie* para gerar uma lista com funções de agregação no pedido, e que a soma do pedido ultrapasse 5000 reais. Com esta lista de códigos de cliente criamos um *where* que retorna os dados da tabela de clientes.

6 Transações

Um importante recurso dos bancos de dados mais recentes é chamado de transação.

Transação consiste em criar um processo atômico onde todas as tarefas devem ou não ser realizadas por completo.

O processo das transações tecnicamente são processos onde as alterações, inclusões e exclusões de linhas da tabela não são gravadas fisicamente, mas são geradas em memória, permitindo que ao final da transações possamos escolher gravar ou descartar, similar ao que acontece quando se abre um documento em um editor de texto e ao final podemos gravar ou não as alterações realizadas.

Por exemplo, uma transação bancária sempre consiste em um débito e um crédito. Se no momento do crédito ocorrer um problema o débito também não pode ser gravado. Este processo é também chamado tecnicamente de *unit-of-work (uow)*, ou unidade de trabalho.

6.1 BEGIN...COMMIT...ROLLBACK

Veja abaixo um exemplo onde a transação foi aberta (*begin transaction*) e finalizada com sucesso (*commit transaction*) ou com falha (*rollback transaction*).

```
BEGIN TRAN          --Abre a transação
UPDATE MEMBER
SET lastname='SILVA',
    Firstname='JOAO'
ROLLBACK TRAN  --TERMINA E "CANCELA" AS ALTERACOES
COMMIT TRAN    --TERMINA E GRAVA AS ALTERACOES
SELECT * FROM MEMBER
```

Note que no exemplo acima constam os dois comandos de finalização, mas apenas um pode ser executado a cada transação, não sendo possível reversão.

6.2 LOCKs

Transações geram *lock* nos dados alterados. Isso é necessário uma vez que durante uma transação os dados não estão sendo fisicamente atualizados na tabela, então se outro usuário tentar consultar ou atualizar dados, as alterações ainda não estão gravadas.

Para evitar surgem em cena os locks, onde o banco de dados não permite que outros usuários utilizem aquelas linhas enquanto não ocorrer um *commit* ou *rollback*.

Portanto, é importante que as transações não sejam muito extensa, pois durante o tempo da transação qualquer registro que for alterado estará travado, o que pode travar aplicações que tentem utilizar ou consultar algum dos dados que estão dentro da transação que foi aberta.

DICA: Transações devem ser abertas o mais tarde possível e finalizadas o mais rápido possível.

7 Inserindo Dados

A inserção de dados em bancos de dados relacionais se faz com o comando *insert*, que possui a seguinte sintaxe básica:

```
INSERT tabela(lista de colunas) VALUES(lista de valores)
```

A lista de colunas logo após o nome da tabela é opcional, se forem informadas as colunas chamamos de *insert* referencial e se não forem informadas as colunas de *insert* posicional.

A vantagem do modelo referencial sobre o modelo posicional, é que neste ultimo todas as colunas da tabela precisam constar no *values* para encaixar com as colunas, além de correremos o risco de alterarmos a lista de colunas e os dados ficarem invertidos.

Segue abaixo exemplos de modelo referencial e posicional:

```
--INSERT REFERENCIAL (COLUNAS NOMEADAS)
INSERT MEMBER(LASTNAME,FIRSTNAME,MIDDLEINITIAL,PHOTOGRAPH)
VALUES('SILVA','JOAO','S',NULL)
INSERT MEMBER(LASTNAME,FIRSTNAME)
VALUES('SILVA','JOAO')
--INSERT POSICIONAL (COLUNAS NAO NOMEADAS)
INSERT MEMBER
VALUES('SOUZA','JOAQUIM','B',NULL)
```

Note que no segundo *insert* foi utilizado o modelo referencial com a citação de apenas duas das quatro colunas e mesmo assim o comando executaria com sucesso. Já no exemplo posicional, mesmo as colunas que não terão dados, como a quarta coluna, precisam ser informadas.

7.1 INSERT com SELECT

Ao invés de fazer um *insert* linha a linha podemos importar dados de uma tabela e inserir em outra utilizando o *insert* em conjunto com o *select*.

Veja no exemplo abaixo como inserimos na tabela *member* os registros da tabela *authors*, onde os autores foram inseridos na tabela de membros.

```
--INSERT COM SELECT DE ORIGEM
INSERT MEMBER(LASTNAME,FIRSTNAME,MIDDLEINITIAL)
SELECT AU_LNAME, AU_FNAME, "
FROM PUBS..AUTHORS
```

Nestes casos a sintaxe altera, uma vez que no lugar da cláusula *values* é utilizado uma instrução *select* que fará o papel de gerar a lista de valores, normalmente colocada no *values*.

8 Deletando Dados

A sintaxe básica do comando de deleção é:

```
DELETE tabela WHERE condição
```

Apesar da cláusula *where* constar no *delete* ela é opcional, mas sempre deve ser colocada.

O comando abaixo foi criado sem *where*, portanto ele irá deletar todos os registros da tabela de membros:

```
DELETE MEMBER
```

O exemplo abaixo é o recomendado, pois inclui a definição para o filtro de exclusão:

```
DELETE MEMBER  
WHERE LASTNAME='SILVA' AND FIRSTNAME='JOAO'
```

Um conselho útil na construção de *delete* é utilizar o *select* e depois apenas substituir um pelo outro. Veja como um comando *select* se transforma em um *delete*:

```
SELECT *  
FROM MEMBER AS M  
INNER JOIN PUBS..AUTHORS AS A  
ON M.LASTNAME = A.AU_LNAME AND  
M.FIRSTNAME = A.AU_FNAME  
DELETE  
FROM MEMBER AS M  
INNER JOIN PUBS..AUTHORS AS A  
ON M.LASTNAME = A.AU_LNAME AND  
M.FIRSTNAME = A.AU_FNAME
```

Note que bastou tirar uma linha com a instruções *select* e colocar o *delete* para que funcionasse como o desejado. Ainda podemos notar neste exemplo que ao deletar dados de uma tabela podemos estar fazendo *join* com outras tabelas, onde o resultado do *join* será deletado na primeira tabela do *from*.

DICA: Este teste com select sempre deve ser efetuado para verificar o que será deletado antes de efetuar as exclusões para testar a funcionalidade.

9 Alterando Dados

A sintaxe básica do comando *update* é:

```
UPDATE tabela
SET coluna=valor, coluna=valor, etc...
WHERE condição
```

Assim como o *delete* o comando *update* não tem o *where* obrigatório, mas caso não seja colocado afetará todas as linhas da tabela.

Veja abaixo alguns exemplo de *update* simples e *update* com base em uma tabela relacionada:

```
--UPDATE SIMPLES
UPDATE MEMBER
SET LASTNAME='SILVA',
    FIRSTNAME='JOAQUIM'
WHERE MIDDLEINITIAL IS NULL

--UPDATE COM COLUNAS DE OUTRA TABELA
UPDATE MEMBER
SET MIDDLEINITIAL = SUBSTRING(A.AU_LNAME,1,1)
--SELECT *
FROM MEMBER AS M
INNER JOIN PUBS.AUTHORS AS A
ON M.LASTNAME = A.AU_LNAME AND
    M.FIRSTNAME = A.AU_FNAME
```

Podemos notar no primeiro exemplo que foram alterado para “silva” todos os membros com o nome do meio não preenchido.

No segundo exemplo notamos o mesmo exercício anterior, onde antes da execução do *update* foi feito um teste com *select* para verificar o retorno e só depois o *update*.

10 Índices

10.1 Estrutura de Índices

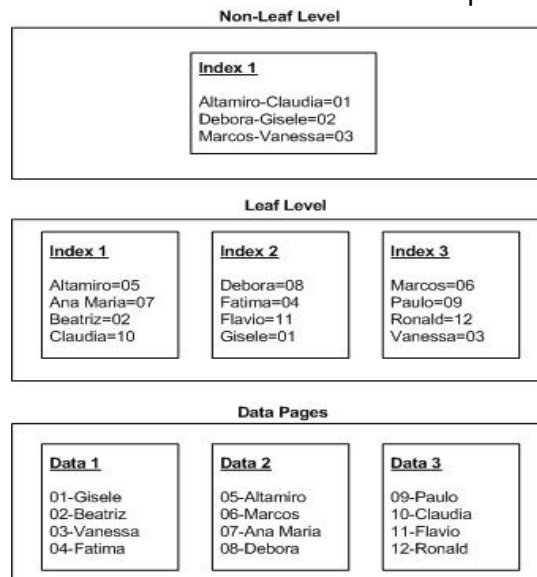
Índices são estruturados sobre os dados da tabela (*data pages*), criando índices (*non-leaf*) e subíndices (*leaf*).

Esta estrutura é similar a estrutura do índice de um livro, ou como desta apostila. A cada módulo, que é o índice de primeiro nível, temos índices de segundo nível.

Ao índice de primeiro nível chamamos de *non-leaf* uma vez que ele não reflete todos as páginas, apenas aponta para um lugar que contem o detalhamento. Já o índice de segundo nível reflete chamamos de *leaf* e ele contem todos os dados ordenados.

O nível *leaf* não contem todos os dados da tabela, mas sim a coluna indexada.

Veja no exemplo abaixo como o banco de dados utiliza o índice para fazer pesquisas.



Caso procure pelo nome Cláudia terá o seguinte processo:

1. Procura no non-leaf, Cláudia está na página leaf numero 1
2. Na página 2 do leaf encontramos Cláudia na linha da tabela número 10
3. Lê os dados da linha 10

Se o índice não existisse seria necessário ter lido todos os 10 registros até chegar na linha desejada, na décima posição.

10.2 Otimizando Performance

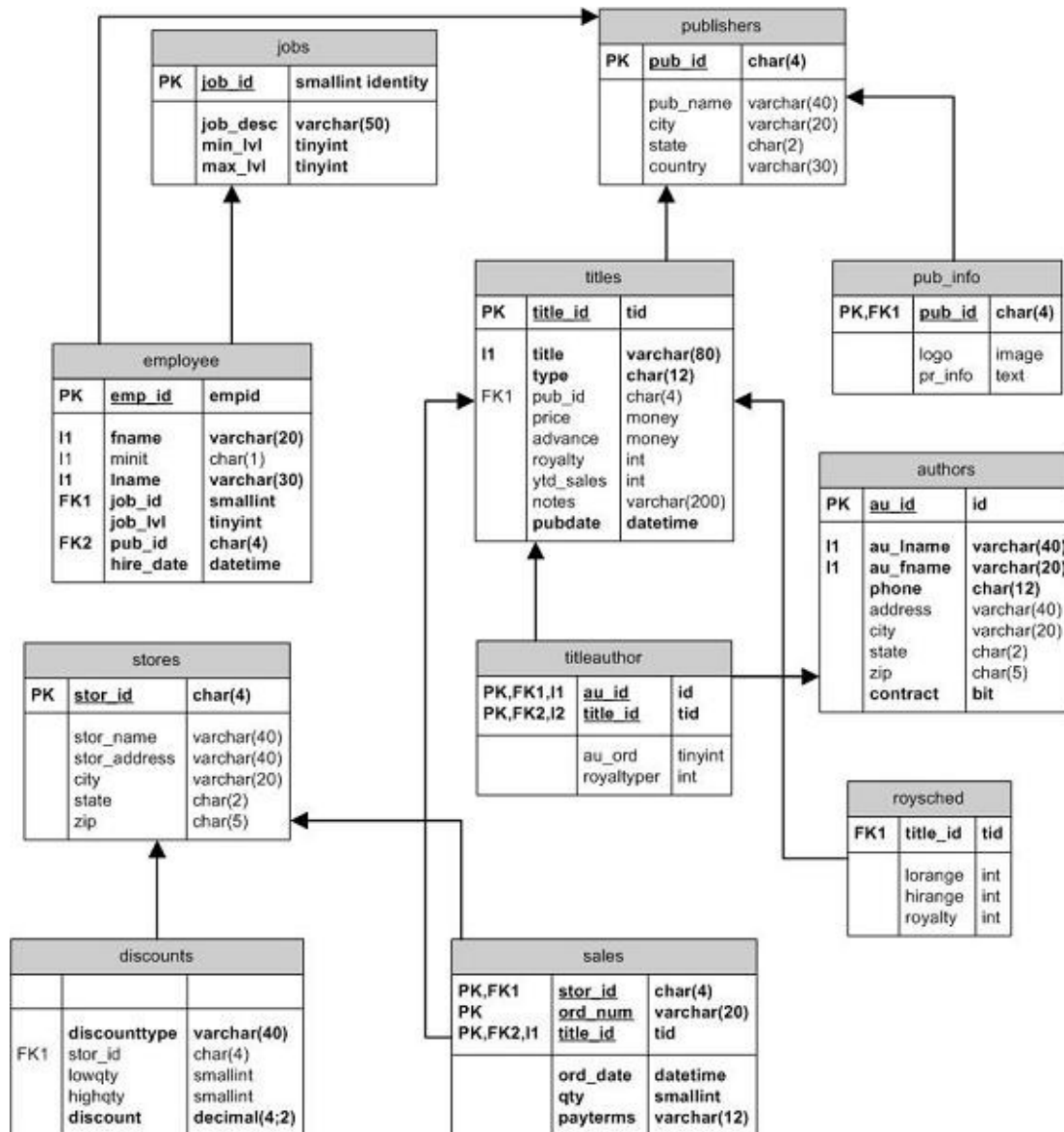
Acima foi feita uma comparação entre uma pesquisa que 10 processos foram substituídos por apenas 3 para encontrar um dado desejado. Agora podemos ter uma idéia da pesquisa ser realizada em uma tabela com cem mil registros.

Ao fazer consultas utilizando o *where*, *order* ou *join* procure utilizar sempre as colunas da tabela que possuem índices.

Aproveitando os índices da tabela podemos ter uma performance, em alguns casos, até 300% mais rápida que consultas sem índices.

11 Diagramas

11.1 PUBS



11.2 Northwind

