2021

INTRODUÇÃO AO SQL

COM MS-SQL SERVER

JAIME H ROZO

jhrozo@yahoo.com

INTRODUÇÃO AO SQL
com MS-SQL Server

Sumário

1. BANCOS DE DADOS

Antes de definir o que é um banco de dados, devemos ter claro que um **dado é um fato ou tipo de informação bruta**. Um dado sozinho não faz sentido.

A **informação** consiste no processo de **agrupar dados de forma organizada para fazer sentido**.

Algumas definições válidas de banco de dados:

> “coleção organizada de dados“;
>
> “coleções de dados interligados entre si e organizados para fornecer informações”;
>
> “coleção de dados inter-relacionados, representando informações sobre um domínio específico”.

Há mais de duas décadas os bancos de dados gerenciados por software se tornaram a principal peça dos sistemas de informação e segurança.

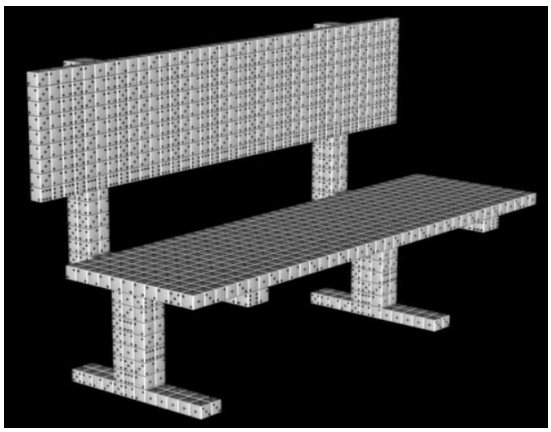
Antes da Era da Informação, os dados eram coletados e armazenados principalmente em cartões, pastas ou em livros contábeis. No caso da contabilidade, os escritórios empregavam pessoas que passavam o dia todo adicionando colunas de números e verificando duas vezes a matemática dos outros, eram os “computadores humanos”.

À medida que os números começaram a ser tratados por máquinas digitais, novas atividades e especializações surgiram. Analistas, programadores, gerentes e funcionários de TI substituíram os "computadores humanos”.

O controle e gerenciamento dos bancos de dados é feito pelo Sistema Gerenciador de Banco de Dados (SGBD). Quando adquirimos um banco de dados estamos adquirindo um software SGBD.

Um sistema de banco de dados consiste num conjunto de quatro componentes básicos: dados, hardware, software e usuários.

O banco feito de dados na figura abaixo poderá ser objeto de um outro manual.



1.1 BENEFÍCIOS DE UM BANCO DE DADOS

■ Promove a consistência de dados.
■ Agiliza a aplicação das regras do negócio.
■ Amplia a capacidade de compartilhamento de dados localmente e a distância.
■ Aprimora a pesquisa e o uso das informações.
■ Facilita a geração de relatórios.
■ Aprimora a capacidade de análise de tendências.

1.2 BANCO DE DADOS RELACIONAIS

Podemos disser que existem 2 grandes categorias de bancos de dados: os bancos de dados relacionais e os não relacionais ou NoSQL.

Grande parte dos sistemas usados por muitas empresas tais como: ERP, CRM e WMS fazem uso dos bancos dados relacionais. Destacam-se os seguintes fornecedores de bancos de dados ou SGBDs: Oracle, IBM e Micosoft com os produtos: Oracle BD, DB2 e SQL Server respectivamente.

A categoria dos bancos não relacionais engloba os seguintes tipos: Document Database (MongoDB), Graph Database (Neo4j), Hierárquico (IBM IMS) e Time Series (Influx DB).

Cada um dos tipos de banco de dados mencionados tem caraterísticas únicas que os habilitam para usos específicos. Na última página listei alguns links sobre os tipos de bancos de dados.

O funcionamento dos bancos de dados relacionais está baseado num modelo proposto por Edgar Codd em 1970 para representar os dados. Em seu trabalho, Codd mostrou que era possível uma visão relacional dos dados para permitir sua descrição de uma forma mais natural, sem estruturas adicionais para sua representação, provendo maior independência dos dados em relação aos programas, sentando assim as bases para tratar problemas como redundância e consistência dos dados. Codd também definiu a álgebra relacional sobre a qual é fundamentada toda a teoria do modelo relacional.

2. O MODELO RELACIONAL

O princípio do modelo relacional é que as informações podem ser consideradas como relações matemáticas baseadas na álgebra relacional. Os conceitos de conjunto, produto cartesiano, seleção, junção, união, subtração e projeção são os principais conceitos usados no modelo relacional.

Os protagonistas do modelo relacional são: as **entidades** e os **relacionamentos**.

Uma **entidade pode ser qualquer coisa** da qual possamos identificar suas caraterísticas e/ou comportamentos, por exemplo: uma pessoa, um carro, um estudante, a mercadoria estocada, um planeta, um cliente etc.

Os **atributos de uma entidade** são todas ou algumas das caraterísticas e/ou comportamentos que identificam ou ajudam a identificar a entidade que estamos observando. Para exemplificar, analisemos a entidade **Cliente**.

Um **Cliente** pode ser: uma pessoa, uma empresa ou um departamento dentro da empresa.

Alguns dos atributos interessantes da entidade **Cliente** podem ser:

- Nome

- Número de identificação (cnpj, cpf, número do departamento etc.)

- Cidade

- País

- Telefone

Outros atributos que podem nos interessar sobre a entidade **Cliente** podem ser:

- Atividade

- E-mail

- Nome do contato

- Logradouro

- Valor Total das Compras

- Data da última compra

O número de atributos de uma entidade tende a ser fixo, mas nada impede que outros possam ser acrescentados. A tarefa de acrescentar um ou mais atributos requer cuidado e planejamento. O nível de dificuldade e esforço tende a ser maior nos bancos de dados relacionais.

A representação gráfica de uma entidade e seus atributos é uma tabela. Cada linha da tabela conterá os **atributos da entidade**, no nosso caso, cada linha conterá os atributos de cada um dos clientes da empresa. O termo **tupla** é a forma de chamar uma **linha ou instância** da entidade no jargão relacional. Como pode notar na figura 1 abaixo, **cada atributo corresponde a uma coluna**.

Entidade

	Atributo 1	Atributo 2	Atributo 3	Atributo 4	Atributo 5
Instância ou tupla 1 >>					
Instância ou tupla 2 >>					
Instância ou tupla 3 >>					

Fig.1 - Representação gráfica de uma entidade com as tuplas vazias.

Considerando a entidade **Cliente** e a lista dos atributos interessantes listados, segue um exemplo gráfico dessa entidade com 3 instâncias ou tuplas:

Entidade: **Cliente**

	Nome	Identificação	Cidade	País	Telefone
Instância ou tupla 1 >>	Jair Silva	102.321.644-99	Manaus	Brasil	(92) 1234-5678
Instância ou tupla 2 >>	Cia & Cia	00.909.999/0001-99	São Paulo	Brasil	(11) 9876-5432
Instância ou tupla 3 >>	Almoxarifado	ALMX	Campinas	Brasil	(19) 3454-9087

Fig.2 – A entidade Cliente com 3 instâncias.

O número de linhas ou tuplas numa tabela é denominado **cardinalidade**. A entidade **Cliente** acima possui **cardinalidade 3** e pode variar para mais ou para menos

O número de atributos ou colunas na tabela é denominado **grau**. A entidade **Cliente** acima possui **grau 5** e tende a não variar ao longo do tempo.

2.1 RELACIONAMENTOS

Observando os atributos da tabela **Cliente** é possível identificar onde ele está localizado. No caso, os atributos “Cidade” e “País” nos permitem **relacionar** ou associar o cliente ao local onde ele se encontra. Podemos dar um nome a cada relação, por tanto, vamos chamar a relação Cliente-Cidade-País de **Cliente-Localização**.

CLIENTE

Cliente-Localização

Nome	Identificação	Cidade	País	Telefone
Jair Silva	102.321.644-99	Manaus	Brasil	(92) 1234-5678
Cia & Cia	00.909.999/0001-99	São Paulo	Brasil	(11) 9876-5432
Almoxarifado	ALMX	Campinas	Brasil	(19) 3454-9087

Fig.3 – Relação Cliente-Localização.

Em algum momento um ou mais clientes podem estar presentes em 2 ou mais locais numa mesma cidade ou em várias cidades. Observe a tabela:

CLIENTE

Cliente-Localização

Nome	Identificação	Cidade	País	Telefone
Jair Silva	102.321.644-99	Manaus	Brasil	(92) 1234-5678
Cia & Cia	00.909.999/0001-99	São Paulo	Brasil	(11) 9876-5432
Almoxarifado	ALMX	Campinas	Brasil	(19) 3454-9087
Jair Silva	102.321.644-99	Manaus	Brasil	(92) 8888-5666
Cia & Cia	00.909.999/0001-99	Manaus	Brasil	(92) 9876-5432

Fig.4 – Clientes em vários locais.

No lugar de ficar escrevendo repetidamente o nome da cidade e o país seria interessante substituir as 2 colunas por apenas uma contendo um indicador que aponte diretamente para a cidade e o país onde está localizado o cliente. Para fazer isso são necessários os seguintes passos:

1. criar uma nova tabela para conter os nomes das cidades e países;
2. incluir na nova tabela uma coluna que permita identificar, de forma única, cada linha contendo a cidade e o país;
3. retirar as colunas cidade e país da tabela Cliente;
4. incluir uma coluna na tabela Cliente para informar a identificação dada à cidade e o país do Cliente na nova tabela.

Os passos descritos fazem parte de um processo conhecido como **normalização**.

Com a **normalização** evitamos ambiguidades e redundância de dados nas tabelas e damos clareza ao nosso projeto de banco de dados. Existem vários níveis de normalização.

A **normalização** implica na criação de **outras entidades**.

Para **relacionar 2 ou mais entidades** entre si deve existir **ao menos um atributo em comum** entre elas.

A figura abaixo mostra o resultado da **normalização** da tabela **Cliente**, note a nova tabela **Localização**. A seta azul indica o **relacionamento entre as tabelas** e o nome da relação continua **Cliente-Localização**. O relacionamento se dá através da coluna **Cód_local** que é o atributo que ambas as tabelas tem em comum. É uma boa prática dar o mesmo nome à coluna que permite o relacionamento nas tabelas.

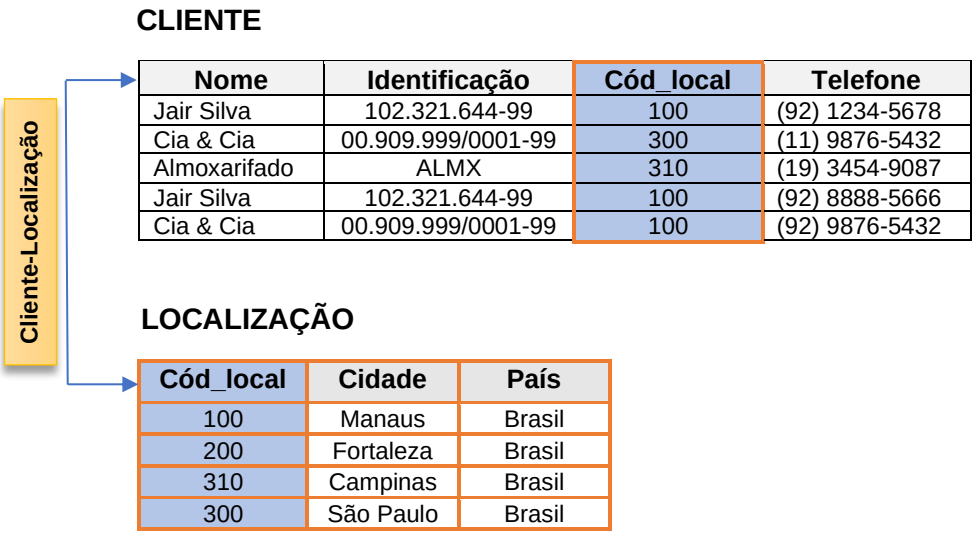


Fig.5 – Resultado da normalização da tabela Cliente.

No jargão relacional, o atributo **Cód_local** é conhecido como **chave**.

Note que o **valor da chave** ele é único na tabela **Localização,** mas ele se repete na tabela **Cliente** tantas vezes quanto linhas ou tuplas existirem nessa tabela para indicar em qual cidade e país um cliente está localizado.

Pelo anterior, podemos disser que **para uma linha na tabela Localização pode não existir nenhuma linha**, **existir apenas uma linha** ou **existirem muitas linhas** na tabela **Cliente**. Em outras palavras, podemos não ter um cliente em determinada cidade e país ou, apenas um cliente em determinada cidade e país, ou muitos clientes em determinada cidade e país. Isto é denominado **cardinalidade do relacionamento**.

Note que a tabela **Localização** pode conter todas as cidades de todos os países do mundo ou de apenas de um país, conforme a necessidade, mas não implica que para cada cidade e país existirá um cliente. Basta entender que esta estrutura poderá suportar tudo isso se for necessário.

É importante destacar que a tabela **Cliente é dependente da tabela Localização** pois é através da tabela **Localização** que é possível saber o nome da cidade e o país onde o cliente está localizado. A dependência é fundamental para interpretar a cardinalidade do relacionamento.

Basicamente, a **cardinalidade de um relacionamento** pode ser:

- de uma linha para outra linha, ou de um para um (1:1),
- de uma linha para muitas linhas, ou de um para muitos (1:n),
- de muitas linhas para muitas linhas, ou de muitos para muitos (m:n) e
- auto relacionamento.

Existem algumas formas gráficas de representar as entidades e os relacionamentos. Foram desenvolvidos softwares especializados para este fim, são muito usados por arquitetos de

sistemas, analistas e programadores de bancos de dados para facilitar a criação do **mapa das entidades e relacionamentos** de um projeto.

A representação gráfica consiste basicamente em retângulos que são conectados entre si por linhas. Os retângulos representam as entidades e as linhas os relacionamentos.

A figura 6 abaixo é a representação gráfica do relacionamento entre as entidades **Cliente** e **Localização** com algumas anotações sobre os elementos que compõem o gráfico:

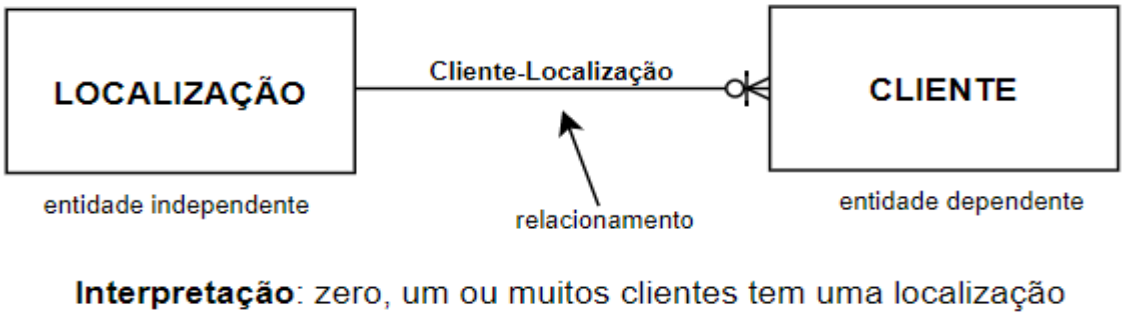


Fig.6 - Representação gráfica das entidades e seu relacionamento.

O gráfico nos indica que a cardinalidade da relação é 1:n, isto é, um-para-muitos, à primeira vista. Isto é sinalizado pelo tipo de simbologia usado em ambos extremos da linha para ligar as duas entidades. Estes símbolos são os **conectores**.

Note o "**pé de galinha**" saindo da entidade Cliente. Este indica que muitas tuplas ou linhas dessa entidade estão relacionadas com uma única tupla ou linha na entidade Localização. A unicidade do lado da entidade Localização é indicada com uma conexão simples.

É possível notar também **uma barra vertical** e **um círculo** depois do "pé de galinha". A barra vertical simboliza o número um e o círculo, o número zero. Isto significa que a relação admite que é possível não ter um cliente ou, ter ao menos, um cliente.

Juntando tudo o anterior, a interpretação do gráfico fica da seguinte maneira: "zero, um ou muitos clientes tem uma localização".

Um exemplo de uma relação 1:1, um-para-um, pode ser entre o código IMEI do celular e uma pessoa física, pois apenas pode existir uma pessoa que possua um celular com um IMEI, o qual é um código único. A figura 7 mostra o diagrama deste relacionamento.

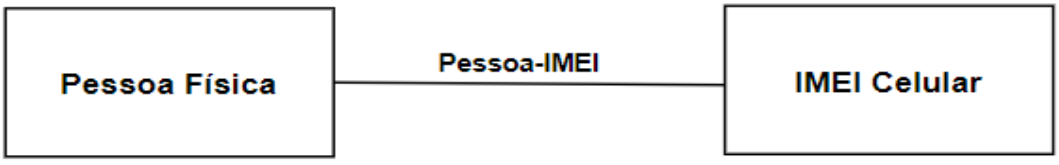


Fig.7 – Diagrama de um relacionamento 1:1.

A relação muitos-para-muitos, n:n, pode ser exemplificada com as entidades Produtos e Pedidos. Muitos produtos podem fazer parte de muitos pedidos. Outro exemplo pode ser entre Cursos e Alunos: cada aluno pode atender muitos cursos e cada curso pode ter muitos alunos.

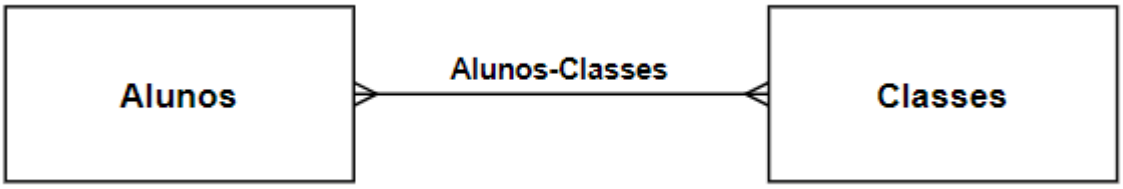


Fig.8 - Representação gráfica do relacionamento n:n.

Nos **diagramas dos modelos de dados** é possível identificar os atributos de cada entidade. A figura 9 mostra o diagrama da relação Cliente-Localização com os atributos.

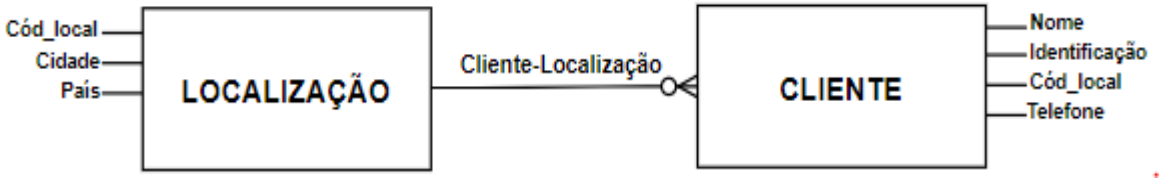


Fig.10 – Representação dos atributos de cada entidade.

Outra forma de visualizar as entidades e os relacionamentos, presente inclusive em muitas ferramentas, é a da figura 11 abaixo. Essa forma admite a inserção de mais detalhes sobre cada entidade e seus atributos os quais foram omitidos intencionalmente.

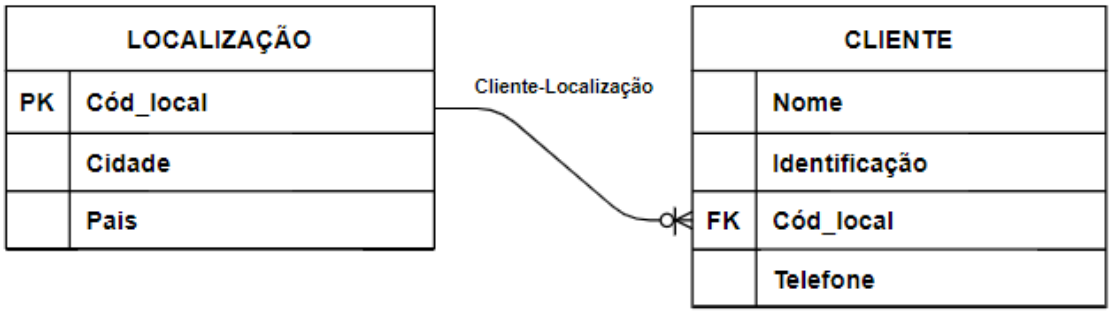


Fig.11 – Outra forma de representar entidades e relacionamentos.

2.2 CHAVES

Uma chave é o valor de um atributo ou a combinação de dois ou mais valores de vários atributos que permitem a identificação de uma linha.

Considerando a tabela **Localização** e o seu conteúdo (veja a figura 5), pode notar que os valores do atributo **Cód_local** não se repetem, são únicos, cada um deles identifica uma única linha. Por esta razão podemos chamar o atributo **Cód_local** de **chave primária** ou **primary key** (**PK**). Somente uma chave primária é permitida por tabela. Caso existam outros atributos cujos valores sejam únicos podemos considera-los como **chaves candidatas** (**CK**).

Observe que os valores do atributo **Cód_local** na tabela **Cliente** se repetem, por isso esse atributo não pode ser uma chave primária, em outras palavras, o atributo **Cód_local** não consegue identificar de forma única cada linha da tabela **Cliente**, ele identifica múltiplas linhas. O valor deste atributo serve apenas para referenciar a chave na tabela **Localização**. Por este motivo é chamado de **chave estrangeira** ou **foreing key** (**FK**). Uma tabela pode ter várias chaves estrangeiras.

Na figura 11 acima podemos ver a nomenclatura PK e FK sendo usada para identificar a chave primária e a chave estrangeira nas entidades **Localização** e **Cliente** respectivamente.

3. SQL – Structured Query Language

A linguagem teve sua origem nos anos 70. Ela fazia parte de uma especificação técnica de um sistema de banco de dados relacional experimental da IBM, o System R. Outras empresas e universidades nos Estados Unidos estavam no processo de pesquisa e desenvolvimento de uma linguagem com as mesmas caraterísticas, era o caso da Oracle e da Universidade de Berkeley.

Devido ao grande interesse e à rápida adoção, foram formados comités para a padronização do SQL pela ANSI (American National Standards Institute) em 1986 e pela ISO (International Organization for Standarization) em 1987. Graças a esses esforços, a primeira especificação padronizada do SQL surgiu em 1989, conhecida como SQL-89. A especificação continua sendo revisada e expandida até hoje.

Cada uma das grandes empresas que hoje oferecem um software gerenciador de bancos de dados adotaram a especificação padrão e ainda, incorporaram funcionalidades específicas desenvolvidas internamente por cada uma delas, conhecidas como extensões. Por este e outros motivos, cada versão comercial possui diferenças funcionais muito específicas, mas sem abandonar os princípios definidos nas especificações definidas pela ANSI e ISO.

Em português, o significado da sigla **SQL é traduzido para**: **Linguagem para Consulta Estruturada**.

Mas a linguagem SQL não é apenas uma linguagem para fazer consultas. Outras funcionalidades ou camadas foram acrescentadas à linguagem ao longo do tempo permitindo mais controle e sofisticação, são elas:

1. **DDL (Data Definition Language)**, permite a criação e manipulação das estruturas que onde são armazenados os dados.
2. **DML (Data Manipulation Language)**, permite a manipulação dos dados dentro das tabelas.
3. **DQL (Data Query Language)**, permite a consulta ou extração de dados das tabelas.
4. **DTL (Data Transaction Language)**, permite o controle de transações.
5. **DCL (Data Control Language)**, permite o controle de usuários, permissões e privilégios.

3.1 Data Query Language (DQL) - Linguagem para Consulta de Dados

É o sub conjunto de comandos da linguagem SQL que permitem a extração das linhas que contém os dados que desejamos extrair das tabelas.

Antes de iniciar o processo de extração é necessário ter claro como os dados estão dispostos no banco de dados, ou seja, você precisa ter uma noção clara de quais estruturas ou tabelas contém os dados e como elas se relacionam entre si e, quais são as chaves primárias (PK) e as chaves estrangeiras (FK). Uma ferramenta que facilita isso é o **Diagrama das Entidades e Relacionamentos**, normalmente ele é fornecido pelo administrador do banco de dados. A maioria dos exemplos de aqui em diante estarão baseados no diagrama mostrado na figura 12 na página seguinte. Ele representa um modelo de dados simples de um controle de pedidos com as seguintes tabelas: Produto, Fornecedor, Cliente, Pedido e ItemPedido.

No diagrama podemos ver que fora da coluna com o nome do atributo temos as colunas “Tipo Condensado” e “Permite Valor Nulo”.

A coluna “Tipo Condensado” serve para identificar o tipo de dado do atributo. Alguns desses tipos são: int (inteiro), decimal (entre parêntesis, o tamanho e o número de casas decimais), nvarchar (string de carateres com tamanho variável), bit (é uma variação do tipo inteiro que só aceita os números 0 ou 1).

A coluna “Permite Valor Nulo” indica a obrigatoriedade de informar um valor para o atributo. A palavra “Não” indica que é obrigatório informar um valor. A palavra “Sim” indica que o atributo não precisa conter um valor. Este tipo de informação é importante para o gerenciador do banco de dados, é um dos parâmetros usados para o controle da consistência dos dados.

Observe o ícone amarelo em formato de chave, ele indica se o atributo é uma chave primaria (PK). O valor da chave primária sempre deve ser informado, por isso a palavra “Não” na coluna “Permite Valor Nulo”.

Note as iniciais “FK” (foreing key ou chave estrangeira) no início do nome de cada relacionamento seguido do nome do atributo que permite a relação, por exemplo: o nome do relacionamento “FK_PEDIDOID_REFERENCIA_PEDIDO” indica que o **atributo PedidoID da tabela ItemPedido é uma chave estrangeira e por tanto, seu valor e o mesmo que o da chave primária (PK) da tabela Pedido**. Os nomes dos relacionamentos no diagrama foram automaticamente gerados pelo SGBD, mas você pode editá-los.

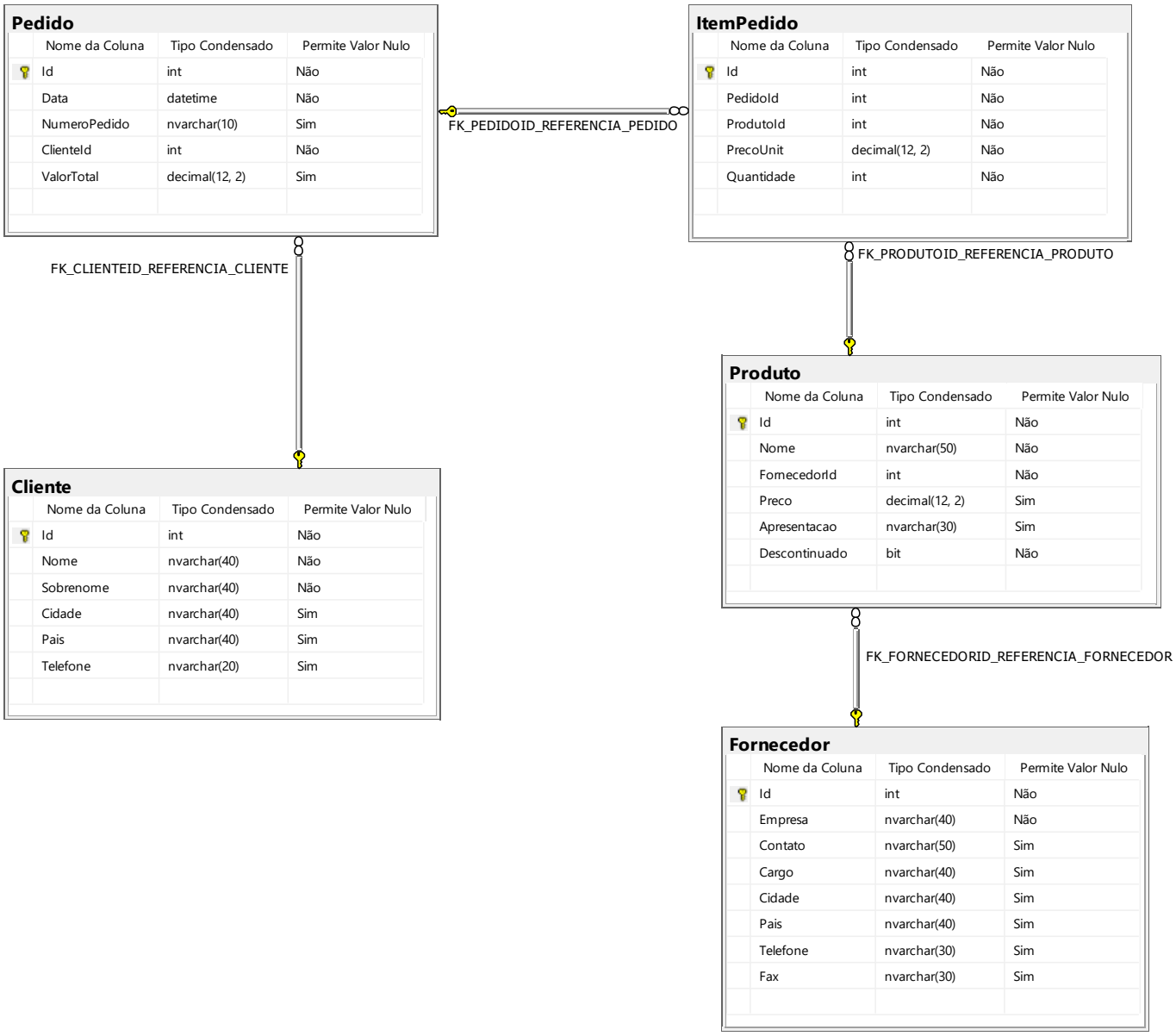


Fig.12 – Diagrama do modelo de dados para um controle de pedidos simples.

3.2 SELECT

Permite **a extração das linhas** de uma ou várias tabelas e **a exibição dos valores** das colunas que nos interessam.

O resultado do SELECT é sempre um conjunto de linhas chamado **result set** (conjunto resultante). A grosso modo, o **result set** é uma tabela criada pelo sistema gerenciador do banco de dados com o resultado da consulta.

A sintaxe básica é:

```
1. SELECT lista_de_colunas
2.  FROM nome_da_tabela
```

onde:

lista_de_colunas: lista com os nomes das colunas cujos valores deseja exibir

```
                    separados por vírgulas.
nome_da_tabela: nome da tabela a consultar.
```

A cláusula **FROM** serve para indicar a origem dos dados. Pode conter o nome de uma ou mais tabelas e, inclusive, uma ou mais cláusulas SELECT, considerando o que foi dito acima sobre o que é um **result set**.

Use um asterisco (*****) ao invés de uma lista com os nomes das colunas para indicar que deseja exibir os valores **de todas as colunas** da tabela:

```
1. SELECT *
2.   FROM nome_da_tabela
```

Exemplos:

```
1. SELECT Id, Nome, Cidade
2.   FROM Cliente
3.
4. SELECT Empresa, Contato, Telefone
5.   FROM Fornecedor
6.
7. SELECT *
8.   FROM Cliente
9.
10. SELECT *
11.   FROM Fornecedor
```

O uso do SELECT nos exemplos acima não é recomendado quando a tabela consultada tiver muitas linhas, a apresentação do resultado pode demorar muito e você terá que forçar a interrupção. Não é incomum encontrar tabelas com centos de miles de linhas em bancos de dados de muitas empresas e de bilhões de linhas em tabelas de empresas de tecnologia da informação como Google.

O uso do asterisco também não é recomendado quando o número de colunas da tabela é muito grande pois dificulta a visualização.

Ao executar os exemplos acima você deve ter percebido que o resultado mostrado não possui uma ordem alfabética nem numérica. Este tipo de extração é chamado de **extração natural** e o gerenciador do banco de dados **não garante nenhuma ordenação do resultado**.

É possível executar um SELECT sem a cláusula FROM conforme os exemplos abaixo:

```
1.  SELECT 2 + 2
2.
3.  SELECT GETDATE()
4.
5.  SELECT UPPER('abcd efg')
6.
7.  SELECT 'Curso de SQL'
```

3.3 CLÁUSULA TOP - LIMITANDO O NÚMERO DE LINHAS A EXIBIR

A cláusula TOP limita o número de linhas que devem ser mostradas.

Nos exemplos abaixo serão selecionadas as 10 primeiras linhas da tabela Cliente e as 30 primeiras da tabela Fornecedor:

```
1.  SELECT TOP(10) Id, Nome, Cidade
2.    FROM Cliente
3.
4.  SELECT TOP(30) Empresa, Contato, Telefone
5.    FROM Fornecedor
```

3.4 USO DO ALIAS

Você pode usar o recurso do **alias** para **substituir o nome de uma tabela**. Para definir o alias de uma tabela use a cláusula **AS** após o nome da tabela dentro da cláusula **FROM**.

O **alias** permite que nomes grandes ou complexos possam ser substituídos em instruções extensas de forma eficiente. Veja o exemplo abaixo:

```
1.  SELECT c.Id,
2.         c.Nome,
3.         c.Cidade
4.    FROM Cliente AS c
5.
6.  SELECT f.*          -- lista todas as colunas da tabela Fornecedor
7.    FROM Fornecedor AS f
```

Em algumas ocasiões, o nome de uma coluna é o mesmo em duas ou mais tabelas que estão sendo usadas numa consulta. Para evitar o conflito com os nomes a solução é criar um **alias** para cada tabela e prefixar o nome da coluna com o alias:

```
1.  SELECT c.Id, c.Nome, c.Cidade,     -- dados da tabela Cliente, alias c
2.         f.Id, f.Empresa, f.Contato      -- dados da tabela Fornecedor, alias f
3.    FROM Cliente AS c, Fornecedor AS f
```

3.5 MUDANDO O NOME DAS COLUNAS NA LISTAGEM

Você pode alterar o nome da coluna no cabeçalho da listagem de saída usando a cláusula **AS** permitindo assim mais clareza na descrição. Veja o exemplo abaixo as duas formas de informar um nome para as colunas:

```
1.  SELECT PedidoId AS Número_Pedido,
2.         ProdutoId AS Código_Produto
3.   FROM ItemPedido
4.
5.  SELECT PedidoId AS [Número Pedido],
6.         ProdutoId AS [Código Produto]
7.   FROM ItemPedido
```

Note a diferencia na assinação do nome das colunas nos dois exemplos acima. Se deseja usar espaços no nome da coluna, coloque alias entre "[" e "]", estes não serão impressos no cabeçalho.

3.6 COLUNAS CALCULADAS

A cláusula **SELECT** admite expressões que resultem no cálculo de um valor ou de uma sequência de carateres (string) na lista de colunas. As expressões podem conter qualquer tipo de função nativa do gerenciador do banco de dados

No primeiro exemplo abaixo, o alias **ValorTotal** foi usado para identificar o cálculo do custo de cada item da tabela **ItemPedido**.

No segundo exemplo, a função **UPPER()** permite converter todas as letras do nome do cliente em maiúsculas.

```
1.  SELECT PedidoId as Número_Pedido,
2.         ProdutoId as Código_Produto,
3.         PrecoUnit * Quantidade as ValorTotal   -- coluna calculada
4.   FROM ItemPedido
5.
6.  SELECT c.Id as Cód_Cliente,
7.         UPPER(c.Nome),     -- mostra o nome com letras maiúsculas
8.         c.Cidade
9.   FROM Cliente as c
```

3.7 DISTINCT – OMITINDO AS REPETIÇÕES

Esta cláusula permite listar uma linha com os valores de um ou mais atributos que não se repetem nas linhas seguintes.

Considere a coluna **Cidade** na tabela **Cliente**. Ela contém o nome das cidades onde os clientes estão localizados e, muitos clientes podem ter endereço numa mesma cidade, por tanto, o nome de uma mesma cidade deve constar em múltiplas linhas. Para obter a lista de todas as cidades onde estão localizados os clientes omitindo as repetições execute a consulta abaixo:

```
1. SELECT DISTINCT Cidade
2.  FROM Cliente
```

3.8 ORDER BY

Permite fazer a ordenação do resultado.

ORDER BY aceita o nome de uma ou mais colunas como critério de ordenação. É possível informar se a sequência da ordenação é no sentido ascendente ou descendente usando os parâmetros **ASC** para ascendente (default) ou **DESC** para descendente.

O exemplo abaixo produz uma lista com o sobrenome, nome e cidade dos clientes, sendo que o sobrenome e o nome estão ordenados alfabeticamente em ordem ascendente graças a cláusula **ASC**. A sequência ascendente é o padrão, por tanto, a cláusula **ASC** não precisa ser especificada. Já o nome da cidade será ordenado na sequência descendente (**DESC**).

```
1. SELECT Sobrenome, Nome, Cidade
2.  FROM Cliente
3. ORDER BY Sobrenome ASC, Nome ASC, Cidade DESC
```

3.9 WHERE

A cláusula WHERE permite informar os critérios ou condições para filtragem das linhas. Somente as linhas que cumprem todas as condições informadas serão extraídas. O resultado de cada condição pode ser ou verdadeiro ou falso, semelhante ao IF das linguagens de programação.

É possível usar os operadores **AND** e **OR**, e os operadores lógicos **=**, **!=**, **<**, **<=**, **>**, **>=**. O operador lógico **!=** (não igual) é equivalente ao **<>** (diferente de).

```
1. SELECT Sobrenome, Nome, Cidade
2.  FROM Cliente
3. WHERE Cidade = 'Rio de Janeiro'
4. ORDER BY Sobrenome ASC, Nome ASC
5.
6. SELECT NumeroPedido, ClienteID, ValorTotal
7.  FROM Pedido
8. WHERE ValorTotal >= 400 AND ValorTotal < 700
9.
10. SELECT Sobrenome, Nome, Cidade
11.  FROM Cliente
12. WHERE Cidade != 'Rio de Janeiro'
13. ORDER BY Sobrenome, Nome, Cidade DESC
```

3.10 NULL - AUSÊNCIA DE VALOR

Quando um atributo não possui um valor dissemos que esse atributo está vazio. A linguagem SQL adotou a palavra **NULL** para identificar a ausência de valor num atributo. A ausência de valor num atributo pode acontecer por diversos motivos que podem ser intencionais ou não, especificamente naqueles em que não é obrigatório informar um valor (veja a coluna “Permite Valor Nulo” nas tabelas do diagrama do modelo de dados acima). Por exemplo: o número do fax do **Fornecedor** não é obrigatório de acordo com o diagrama, por tanto, pode acontecer que nem todos serão informados, ou talvez, que nenhum seja informado.

Não confunda a ausência de valor com espaços. Um espaço é considerado um carater assim como uma letra ou um número. A forma de verificar se o valor de um atributo é composto por espaços é diferente da forma de verificar se o atributo possui um valor.

Continuando com o exemplo do atributo **Fax** na tabela **Fornecedor**, podemos listar todos os fornecedores que não tem número de fax usando o operador **IS NULL** como filtro na cláusula **WHERE**.

Para obter uma lista dos fornecedores com número de fax use o operador **IS NOT NULL**.

```
1. SELECT Empresa, Fax   -- lista fornecedores sem o número do fax
2.  FROM Fornecedor
3.  WHERE Fax IS NULL
4.
5. SELECT Empresa, Fax   -- lista fornecedores com o número do fax
6.  FROM Fornecedor
7.  WHERE Fax IS NOT NULL
```

3.11 GROUP BY – GRUPOS DE LINHAS

Permite a divisão do conjunto de linhas do resultado da consulta em grupos baseado no valor de um atributo comum entre elas. Esta cláusula é muito usada para a sub totalização e totalização. A cláusula SELECT retornará uma a uma as linhas de cada grupo. Esta cláusula não garante qualquer tipo de ordenação dos grupos.

Suponha que alguém deseja saber o número de clientes por cidade, a consulta ficaria da seguinte forma:

```
1. SELECT Cidade, COUNT(*) as Contagem
2.  FROM Cliente
3.  GROUP BY Cidade
```

A função **COUNT(*)** é uma das muitas funções nativas dos SGBDs. Ela permite contar o número de vezes que o valor de um atributo está presente numa tabela. O asterisco indica que é para considerar todas as linhas na contagem mesmo que o atributo não possua um valor, isto é, seja NULL.

O resultado da consulta acima é uma listagem contendo o nome de cada cidade e a contagem das vezes, ou de linhas, que cada cidade aparece (“Contagem”).

Podemos melhorar a apresentação da consulta acima organizando a contagem em ordem descendente, veja o primeiro exemplo abaixo. O segundo exemplo mostra a contagem de todas as cidades de cada país.

Se por acaso algumas linhas no tiverem um valor na coluna informada como parâmetro do GROUP BY, estas linhas serão listadas devidamente num grupo que será automaticamente denominado "NULL". Execute o terceiro exemplo abaixo para saber quantos fornecedores não tem um número de fax.

```
1.  SELECT Cidade, COUNT(*) as Contagem
2.   FROM Cliente
3.  GROUP BY Cidade
4.  ORDER BY Contagem DESC   -- usando o nome assinado à coluna
5.
6.  SELECT Pais, Cidade, COUNT(*) as Contagem
7.   FROM Cliente
8.  GROUP BY Pais, Cidade
9.  ORDER BY Pais, Cidade
10.
11. SELECT Fax, COUNT(*) as Contagem
12.  FROM Fornecedor
13. GROUP BY Fax
14. ORDER BY Contagem DESC
```

3.12 HAVING

Serve para filtrar os grupos formados com a cláusula GROUP BY. O funcionamento é semelhante à cláusula WHERE na medida que atua com filtro no processamento dos grupos.

A consulta abaixo extrai todos os países que tenham um número de cidades maior que 2:

```
1.  SELECT Pais, Cidade, COUNT(*) as Contagem
2.   FROM Cliente
3.  GROUP BY Pais, Cidade
4.  HAVING COUNT(*) > 2     -- não pode ser usado o nome "Contagem"
5.  ORDER BY Pais, Cidade
```

3.13 OPERADORES DE CONJUNTOS: UNION, UNION ALL, INTERSECT, EXCEPT

Um operador de conjunto permite combinar os resultados de duas ou mais consultas numa só.

3.13.1 UNION e UNION ALL

Permite a combinação dos resultados de duas ou mais consultas em um **único result set**. Por definição, o operador **UNION elimina as duplicações**.

O número de colunas e o tipo de cada coluna na cláusula SELECT de cada consulta tem que ser os mesmos.

```
1. SELECT 'Cliente', Nome, Cidade, Pais
2.   FROM Cliente
3.  UNION
4. SELECT 'Fornecedor', Empresa, Cidade, Pais
5.   FROM Fornecedor
6.  ORDER BY Pais, Cidade, Nome
```

A consulta acima resulta numa lista ordenada com todos os clientes e fornecedores existentes em cada cidade de cada país. Foram usadas as palavras 'Cliente' e 'Fornecedor' para identificar de quem são os dados em cada linha. Note que o número de colunas é o mesmo em ambos SELECT. Confira no diagrama das entidades, coluna "Tipo Condensado", o tipo de cada atributo que aparece nas duas cláusulas SELECT, tem que ser o mesmo.

UNION ALL funciona da mesma maneira que o operador UNION, a diferença é que o UNION ALL **não elimina as duplicações**

3.13.2 INTERSECT

Retorna um conjunto de linhas resultantes de ambas consultas. As linhas são únicas, isto é, não há repetição de valores entre as colunas de uma linha e outra.

Exemplo: todos os produtos que foram incluídos em algum pedido:

```
1. SELECT Id
2.   FROM Produto
3. INTERSECT
4. SELECT ProdutoId
5.   FROM ItemPedido
```

3.13.3 EXCEPT

Retorna uma lista com de linhas únicas da primeira consulta que não existem na segunda.

Exemplo: os produtos que ainda não foram incluídos em algum pedido:

```
1. SELECT Id
2.   FROM Produto
3. EXCEPT
4. SELECT ProdutoId
5.   FROM ItemPedido
```

4. JOINS

Permitem extrair e juntar dados de duas ou mais tabelas com base em relações lógicas entre as tabelas. Um JOIN indica como devem ser usados os dados de uma tabela para selecionar as linhas de outra tabela.

O JOIN pode ser especificado dentro da cláusula FROM ou WHERE**.** Cada JOIN precisa ter definida a condição da junção, ou seja, o modo como duas tabelas serão relacionadas em uma consulta.

Uma condição de junção típica especifica uma coluna de cada tabela envolvida no JOIN. Normalmente é escolhida a coluna que é uma chave estrangeira em uma tabela e a coluna correspondente à chave primária associada na outra tabela. Depois, é necessário especificar um operador lógico (por exemplo, = ou <>) a ser usado na comparação de valores entre as colunas.

As condições do JOIN combinam-se com as condições das cláusulas WHERE e HAVING para controlar a extração das linhas das tabelas referenciadas na cláusula FROM

Os tipos de JOIN são:

- INNER JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

A palavra OUTER não é obrigatória.

4.1 INNER JOIN

Permite a obter os dados relacionados de duas ou mais tabelas. Lembre-se que um relacionamento entre duas tabelas consiste na existência de uma coluna com valores em comum entre duas tabelas, como é o caso da coluna Id na tabela Cliente e a coluna ClienteId na tabela Pedido, as duas colunas contém um mesmo valor que é o código do cliente, uma delas é chave primária (PK) e a outra é chave estrangeira (FK).

Exemplo: extrair todos pedidos do cliente cujo Id é 63:

```
1. SELECT Nome, Sobrenome, NumeroPedido, Data, ValorTotal
2.  FROM Cliente INNER JOIN Pedido ON Cliente.Id = ClienteId
3. WHERE Cliente.Id = 63
4. ORDER BY Data, NumeroPedido, ProdutoID
```

Podemos ser mais seletivos na consulta acima, vamos extrair os pedidos do mesmo cliente a partir de janeiro 01 de 2014:

```
1. SELECT Nome, Sobrenome, NumeroPedido, Data, ValorTotal
2.  FROM Cliente INNER JOIN Pedido ON Cliente.Id = ClienteId
3. WHERE Cliente.Id = 63
4.    AND Data >= '2014-01-01'
5. ORDER BY Data, NumeroPedido
```

O INNER JOIN pode ser especificado na cláusula WHERE. Usando o mesmo exemplo acima, a consulta fica da seguinte forma:

```
1.  SELECT Nome, Sobrenome, NumeroPedido, Data, ValorTotal
2.   FROM Cliente, Pedido
3.  WHERE Cliente.Id = 63
4.     AND ClienteID = 63
5.     AND Data >= '2014-01-01'
6.  ORDER BY Data, NumeroPedido
```

Consegue entender o porquê de usar Cliente.Id na linha 8? Antes de responder dê uma olhada no diagrama das entidades e relacionamentos

4.2 LEFT JOIN

Permite extrair **todas as linhas da tabela à esquerda do operador JOIN** junto com as linhas da tabela à direita do operador JOIN que tiverem uma relação com a tabela da esquerda. Os valores das colunas da tabela da direita que não puderam ser relacionados são substituídos pela palavra NULL.

A consulta abaixo extrai todos os clientes que fizeram ou não um pedido. Note que para esses clientes aparece NULL nas colunas NumeroPedido, Data e ValorTotal já que não existem linhas na tabela Pedido relacionadas com esses clientes.

```
1.  SELECT c.Id, c.Nome, c.SobreNome, p.NumeroPedido, p.Data, p.ValorTotal
2.   FROM Cliente AS c  LEFT JOIN  Pedido AS p  ON c.Id = p.ClienteId
3.  WHERE p.NumeroPedido IS NULL
4.  ORDER BY p.NumeroPedido
```

4.3 RIGHT JOIN

Permite extrair **todas as linhas da tabela à direita do operador JOIN** junto com as linhas da tabela à esquerda do operador JOIN que tiverem uma relação com a tabela da direita. Os valores das colunas da tabela da esquerda que não puderam ser relacionados são substituídos pela palavra NULL.

Exemplo: consultar quais os produtos que não fazem parte de nenhum pedido:

```
1.  SELECT p.Id, p.Nome, i.PedidoId, i.Quantidade, i.PrecoUnit,
2.             i.Quantidade * i.PrecoUnit AS [Valor]
3.    FROM ItemPedido AS i  RIGHT JOIN  Produto AS p  ON p.Id = i.ProdutoId
4.   WHERE i.PedidoId IS NULL
5.   ORDER BY i.PedidoId
```

4.4 FULL JOIN

Permite extrair **todas as linhas da tabela à esquerda e à direita do operador JOIN**. Os valores das colunas não relacionadas serão substituídos pela palavra NULL.

Exemplo: extrair todos os produtos e todos os pedidos, independentemente de o produto fazer parte de um pedido:

```
1.  SELECT p.Id, p.Nome, i.PedidoId, i.Quantidade, i.PrecoUnit,
2.           i.Quantidade * i.PrecoUnit AS [Valor]
3.    FROM ItemPedido AS i  FULL JOIN  Produto AS p  ON p.Id = i.ProdutoId
4.   ORDER BY i.PedidoId
```

4.5 CROSS JOIN

Permite cruzar todos dados entre a tabela à esquerda e a tabela à direita do operador JOIN gerando um produto cartesiano. É possível cruzar mais de duas tabelas. Este tipo de JOIN deve ser usado com cuidado pois o número de linhas gerado pode ser muito grande. Suponha que duas tabelas possuem 10.000 linhas, o produto cartesiano destas duas tabelas será: 100.000 linhas.

A primeira consulta, no exemplo abaixo, calcula o número de linhas que deverão resultar ao fazer um CROSS JOIN entre as tabelas Cliente e Fornecedor. A segunda consulta mostra todas as linhas do CROSS JOIN. O número de linhas deve ser igual ao calculado na primeira consulta.

```
1.  SELECT COUNT (*) * (SELECT COUNT(*) FROM Fornecedor)
2.   FROM Cliente
3.
4.  SELECT ClienteId, Cliente.Nome,
5.           Fornecedor.Id, Fornecedor.Empresa
6.  FROM Cliente CROSS JOIN Fornecedor
```

Note que não foi necessário especificar a cláusula **ON** no operador CROSS JOIN pois não há necessidade de identificar um relacionamento.

4.6 ORDEM LÓGICA DE EXECUÇÃO DAS CLÁUSULAS DE UMA CONSULTA

A ordem sintática das cláusulas numa consulta difere da ordem como são executadas internamente. O conhecimento da ordem lógica de execução de cada cláusula facilita muito a estruturação das consultas.

A ordem sintática das cláusulas mais comuns é a seguinte:

1. SELECT
2. DISTINCT
3. TOP
4. FROM
5. ON
6. JOIN
7. WHERE
8. GROUP BY
9. WITH CUBE ou WITH ROLLUP
10. HAVING
11. ORDER BY

A ordem lógica de execução é a seguinte:

1. FROM
2. ON

3. JOIN
4. WHERE
5. GROUP BY
6. WITH CUBE ou WITH ROLLUP
7. HAVING
8. SELECT
9. DISTINCT
10. ORDER BY
11. TOP

A ordem lógica determina quando os objetos definidos em uma etapa são disponibilizados às cláusulas em etapas subsequentes. Por exemplo, se o processador de consulta puder acessar as tabelas definidas na cláusula FROM, esses objetos e suas colunas serão disponibilizados para todas as etapas subsequentes. Por outro lado, como a cláusula SELECT é o passo 8, qualquer alias de coluna ou colunas derivadas definidas nessa cláusula não pode ser referenciado por cláusulas anteriores. No entanto, eles podem ser referenciados por cláusulas subsequentes, como a cláusula ORDER BY. A execução física real da instrução é determinada pelo processador de consulta e a ordem pode variar a partir desta lista.

A cláusula FROM é executada primeiro. Ela disponibiliza a fonte dos dados (uma ou múltiplas tabelas) na forma de tabelas virtuais. Em caso das cláusulas ON e JOIN estarem presentes na consulta, serão juntadas todas as colunas das tabelas envolvidas no JOIN numa tabela virtual.

As condições da cláusula WHERE são aplicadas à tabela virtual. O resultado é a tabela virtual sem as linhas que não satisfazem as condições definidas.

A tabela virtual filtrada é passada para a cláusula GROUP BY. O resultado é a tabela virtual organizada em grupos conforme os parâmetros informados. A partir deste momento, cada coluna dos grupos estará disponível para as seguintes cláusulas, com grande impacto na cláusula SELECT.

As cláusulas WITH CUBE ou WITH ROLLUP permitem, em alguns casos, o cálculo de totalizações por grupo e geral. Estão intimamente ligadas à cláusula GROUP BY.

A cláusula HAVING filtra as linhas de cada grupo de acordo com os parâmetros informados.

Finalmente, a cláusula SELECT é processada, determinando quais colunas serão mostradas no resultado da consulta.

5. FUNÇÕES EMBUTIDAS NO SGBD

Todos os SGBDs disponíveis no mercado oferecem um conjunto de funções embutidas

O SQL Server oferece os seguintes tipos de funções:

- **Funções de agregação**: executam um cálculo em um conjunto de valores e retornam um único valor.

- **Funções analíticas**: computam um valor agregado com base em um grupo de linhas. Porém, ao contrário das funções de agregação, as funções analíticas podem retornar várias linhas para cada grupo. Você pode usar funções analíticas para calcular médias móveis, totais acumulados, percentuais ou os primeiros N resultados de um grupo.

- **Funções de classificação**: retornam um valor de classificação para cada linha de um grupo. Dependendo da função usada, algumas linhas podem receber o mesmo valor que outras.

- **Funções de conjuntos**: retornam uma tabela virtual que pode ser usada como referência em uma instrução SQL.

- **Funções escalares**: aceitam um valor e retornam um único valor, semelhante a muitas funções no Excel. Estas serão tratadas a seguir.

5.1 FUNÇÕES ESCALARES

Este grupo de funções é composto por vários subgrupos, alguns deles são:

- **Funções de conversão**: convertem um tipo de dado para outro.
- **Funções de data e hora**: aceitam um valor data/hora e retornam parte ou todo o valor como uma string, um número ou ainda no formato data/hora.
- **Funções lógicas**: permitem a execução de operações lógicas.
- **Funções matemáticas**: aceitam um ou vários valores e retornam um único valor calculado.
- **Funções de strings**: aceitam uma string como parâmetro e retornam outra string ou um valor numérico.

5.2 FUNÇÕES DE CONVERSÃO

- CAST()
- CONVERT()

5.3 FUNÇÕES DE DATA E HORA

- CURRENT_TIMESTAMP
- GETDATE()
- DATENAME()
- DATEPART()
- DAY()
- MONTH()
- YEAR()
- DATEDIFF()
- DATEADD()
- DATEPART()
- EOMONTH()
- ISDATE()

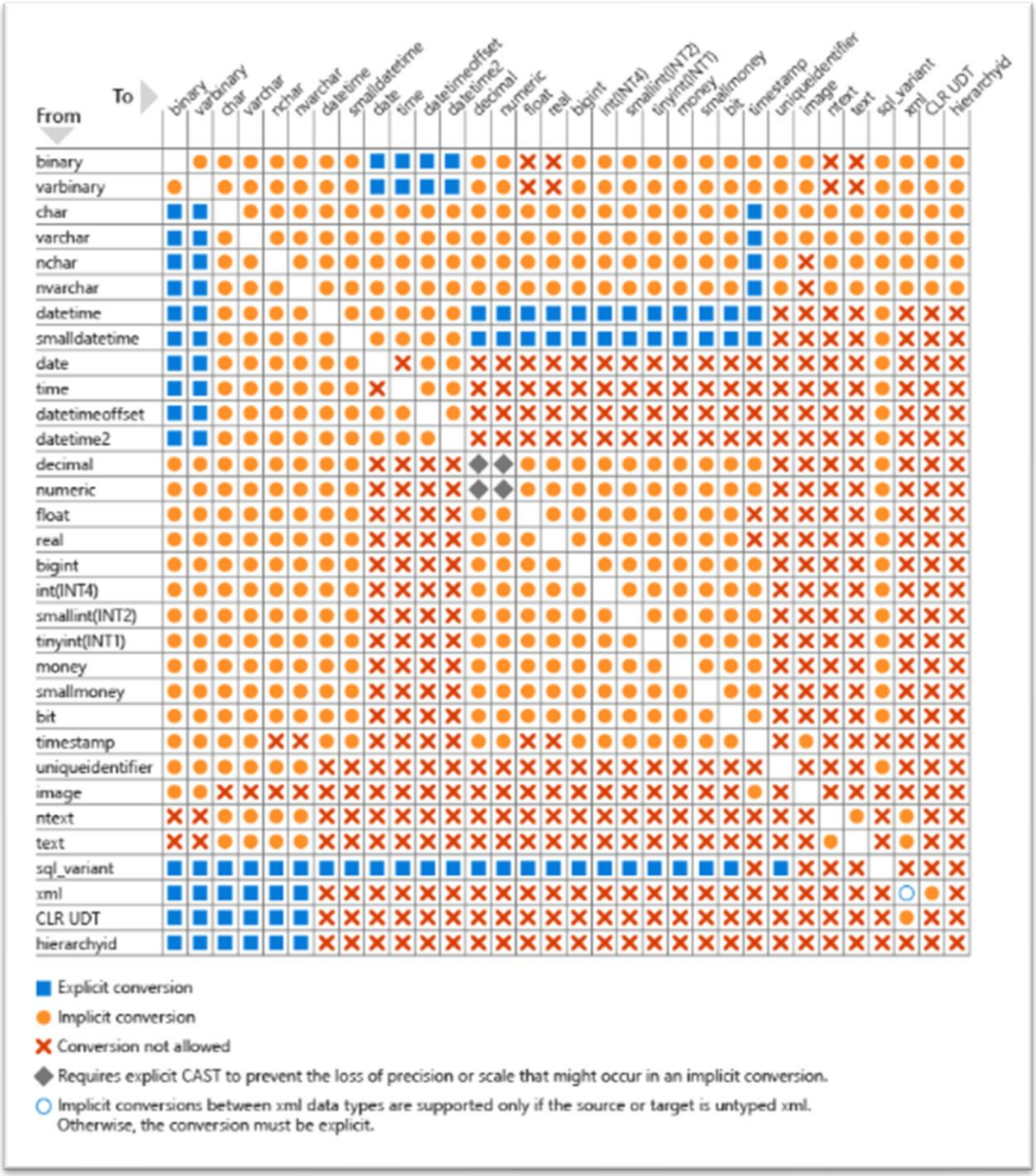


Tabela de conversões implícitas e explícitas permitidas no MS SQL Server

5.4 FUNÇÕES LÓGICAS

- CHOOSE() – retorna o item de uma lista localizado numa posição específica.

5.5 FUNÇÕES MATEMÁTICAS

- ABS()
- ACOS()
- ASIN()
- CEILING()
- EXP()
- FLOOR()

- POWER()
- ROUND()
- SIGN()
- SQRT()

5.6 FUNÇÕES DE STRINGS

- CHAR()
- CHARINDEX()
- CONCAT()
- FORMAT()
- FORMATMESSAGE()
- LEFT()
- LEN()
- LOWER()
- LTRIM()
- REPLACE()
- REPLICATE()
- REVERSE()
- RIGHT()
- RTRIM()
- SPACE()
- STR()
- STRING_AGG()
- STUFF()
- SUBSTRING()
- TRIM()
- UPPER()

5.7 FUNÇÕES DE AGREGAÇÃO

- SUM()
- MIN()
- MAX()
- AVG()
- COUNT()

6. SUBQUERIES

A subquery é uma consulta aninhada numa outra. A consulta que contém uma subquery é chamada de consulta externa (outer query). A query aninhada é chamada de consulta interna (inner query).

Uma subquery pode fornecer seu resultado à consulta externa.

Uma subquery pode retornar um só valor (scalar subquery) ou múltiplos valores (multi-valued subqueries) à consulta externa. Podemos ter também uma subquery referenciando colunas da consulta externa e devolvendo um ou mais valores (correlated subquery). Uma subquery deve estar **sempre entre parêntesis**.

6.1 SCALAR SUBQUERY

Retorna um só valor. Podem ser aninhadas até 32 subqueries dentro de uma consulta.

Pode ser usada em qualquer cláusula que permita o retorno de um só valor (SELECT, WHERE, HAVING ou FROM).

Caso não retorne um valor, o resultado será convertido para NULL. Um tratamento adequado de um resultado NULL deve ser previsto na consulta.

Exemplo: obter os detalhes do último pedido:

```
1. SELECT PedidoId, ProdutoId, PrecoUnit, Quantidade
2.  FROM ItemPedido
3. WHERE PedidoId = (SELECT MAX(Id) AS [Útimo Pedido]
4.                   FROM Pedido)
```

6.2 MULTIVALUED SUBQUERY

Retorna múltiplos valores no formato de uma coluna para a consulta externa. É usada em conjunto com a cláusula **IN**. Pode ser substituída por um JOIN.

Exemplo: listar os produtos dos pedidos feitos entre as datas 04-05-2014 e 06-05-2014:

```
1. SELECT PedidoId, ProdutoId, Quantidade
2.  FROM ItemPedido
3. WHERE PedidoId IN (SELECT Id
4.                    FROM Pedido
5.                   WHERE DATA >= '2014-05-04' AND DATA <= '2014-05-06')
```

A mesma consulta usando JOIN:

```
1.  SELECT PedidoId, ProdutoId, Quantidade
2.  FROM ItemPedido i INNER JOIN Pedido p ON i.PedidoId = p.Id
3. WHERE i.Data >= '2014-05-04' AND i.Data <= '2014-05-06'
4. ORDER BY i.PedidoId, i.ProdutoId
```

6.3 CORRELATED SUBQUERY

São consultas que referenciam colunas das tabelas da consulta externa.

Podem ser lentas na medida que para cada linha retornada pela consulta externa é executada a consulta interna. Podem retornar um único valor ou múltiplos.

Exemplo: listar a data, o número e o valor do último pedido de cada cliente:

```
1.  SELECT ClienteId, Data, NumeroPedido, ValorTotal
2.   FROM Pedido p1
3.  WHERE Data = (SELECT MAX(Data)
4.                  FROM Pedido p2
5.                 WHERE p2.ClienteId = p1.ClienteId)
6.  ORDER BY ClienteId, Data
```

Como a tabela **Pedido** está sendo usada tanto na consulta externa como na interna é necessário assinar um alias à tabela em ambas consultas, no caso, foram assinados os alias **p1** e **p2**. Note o uso do alias para qualificar o nome da mesma coluna na linha 5 para evitar uma ambiguidade na análise e execução do WHERE da parte do SGDB.

SQL Scripts

Faz parte deste manual um conjunto de scripts para criar e popular as tabelas do banco de dados bd_01 usado como modelo para os exemplos. Incluí no conjunto de scripts o arquivo exemplos.sql que contém todos os exemplos deste manual. Para obter todos os scripts e a versão mais atualizada deste manual visite: https://github.com/jhrozo/IntroducaoSQL

Referências

Documentação on-line Microsoft SQL Server

Tipos de Bancos de Dados

Relacional:

1. SQL Server - https://www.microsoft.com/pt-br/sql-server/sql-server-2019
2. Oracle - https://www.oracle.com/br/database/

Não Relacional:

1. Document Data Base

   MongoDB - https://docs.mongodb.com/manual/introduction/

2. Graph Database

   Neo4j - https://neo4j.com/developer/graph-database/

3. Hierárquico

   IBM IMS - https://www.ibm.com/it-infrastructure/z/ims

4. Time Series

   Influx DB - https://www.influxdata.com/