SQL >SQL Introdução

SQL (Linguagem de Consulta Estruturada) é uma linguagem informática destinada a armazenar, manipular e obter dados armazenados em bases de dados relacionais. A primeira encarnação da linguagem SQL apareceu em 1974, quando um grupo dentro da IBM desenvolveu o primeiro protótipo de uma base de dados relacional. A primeira base de dados relacional comercial foi distribuída pela Relational Software (mais tarde passou a chamar-se Oracle).

Existem normas para a linguagem SQL. Contudo, a linguagem SQL que pode ser utilizada em cada um dos principais sistemas RDBMS possui diferentes características. Isso deve-se a duas razões: 1) A norma SQL é relativamente complexa e não é prático implementar toda a norma 2) cada vendedor de bases de dados precisa de formas para diferenciar o seu produto dos outros. Neste tutorial, essas diferenças são mencionadas quando apropriado.

Este site de tutorial de SQL lista os comandos SQL utilizados mais frequentemente e encontra-se dividido nas seguintes secções:

- **Comandos SQL**: Instruções SQL básicas para armazenar, obter e manipular dados numa base de dados relacional.
- **Manipulação de tabelas**: Forma como as instruções SQL são utilizadas para gerir tabelas dentro de uma base de dados.
- **Linguagem SQL avançada**: Comandos SQL avançados.
- **Sintaxe SQL**: Uma única página que lista a sintaxe de todos os comandos SQL neste tutorial.

Para cada comando, a sintaxe SQL será primeira apresentada e explicada, seguindo-se um exemplo. No final deste tutorial, deve possuir uma compreensão geral adequada acerca da sintaxe SQL. Além disso, deve ser capaz de gravar consultas SQL utilizando a sintaxe correta. A minha experiência diz-me que compreender os conceitos básicos da linguagem SQL é muito mais fácil do que dominar todas as complicações desta linguagem de base de dados, e espero que seja capaz de chegar à mesma conclusão.

Se estiver interessado em saber como obter dados através da linguagem SQL, recomendamos que comece pela secção **Comandos SQL**. Se pretender compreender de que forma a linguagem SQL pode ser utilizada para manipular tabelas de bases de dados, recomendamos que comece pela secção **Manipulação de tabelas**. Se pretender obter ajuda sobre um comando SQL específico, pode utilizar o **Mapa do site** para encontrar o comando pretendido.

SQL >SQL Sintaxe

Nesta página, listamos a sintaxe SQL para cada um dos comandos SQL presentes neste tutorial. Para obter explicações detalhadas sobre cada sintaxe SQL, consulte a secção individual clicando na palavra-chave.

O objectivo desta página é obter uma referência rápida acerca da sintaxe SQL. Sugerimos que adicione esta página aos favoritos premindo **Control-D** para que possa aceder rapidamente a esta página de sintaxe.

Select
```
SELECT "nome_coluna" FROM "nome_tabela";
```

Distinct
```
SELECT DISTINCT "nome_coluna"
FROM "nome_tabela";
```

Where
```
SELECT "nome_coluna"
```

```
FROM "nome_tabela"
WHERE "condição";
```

And/Or

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "condição simples"
{[AND|OR] "condição simples"}+;
```

In

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "nome_coluna" IN ('valor1', 'valor2', ...);
```

Between

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "nome_coluna" BETWEEN 'valor1' AND 'valor2';
```

Like

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "nome_coluna" LIKE {PATROON};
```

Order By

```
SELECT "nome_coluna"
FROM "nome_tabela"
[WHERE "condição"]
ORDER BY "nome_coluna" [ASC, DESC];
```

Count

```
SELECT COUNT("nome_coluna")
FROM "nome_tabela";
```

Group By

```
SELECT "nome_coluna 1", SUM("nome_coluna 2")
FROM "nome_tabela"
GROUP BY "nome_coluna 1";
```

Having

```
SELECT "nome_coluna 1", SUM("nome_coluna 2")
FROM "nome_tabela"
GROUP BY "nome_coluna 1"
HAVING (condição da função aritmética);
```

Create Table

```
CREATE TABLE "nome_tabela"
("coluna 1" "tipo_dados_para_coluna_1",
"coluna 2" "tipo_dados_para_coluna_2",
... );
```

Drop Table

```
DROP TABLE "nome_tabela";
```

Truncate Table
TRUNCATE TABLE "nome_tabela";

Insert Into
INSERT INTO "nome_tabela" ("coluna 1", "coluna 2", ...)
VALUES ("valor 1", "valor 2", ...);

Update
UPDATE "nome_tabela"
SET "coluna 1" = [novo valor]
WHERE "condição";

Delete From
DELETE FROM "nome_tabela"
WHERE "condição";

SQL >Comandos SQL >Select

Para que servem os comandos SQL? Uma utilização comum é a seleção de tabelas localizadas numa base de dados. Imediatamente, visualizamos duas palavras-chave: necessitamos de **SELECT** (selecionar) **FROM** (de) uma tabela. (Note que uma tabela é um receptáculo existente numa base de dados onde os dados estão armazenados. Para obter mais informações sobre como manipular tabelas, consulte a secção **Manipulação de tabelas**). Deste modo, temos a estrutura SQL mais básica:

```
SELECT "nome_coluna" FROM "nome_tabela";
```

De modo a ilustrar o exemplo acima apresentado, assuma que possuímos a seguinte tabela:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Iremos utilizar esta tabela como exemplo ao longo deste tutorial (esta tabela irá aparecer em todas as secções). Para selecionar todas as lojas nesta tabela, introduzimos

```
SELECT Store_Name FROM Store_Information;
```

Resultado:

Store_Name
Los Angeles
San Diego
Los Angeles
Boston

É possível selecionar nomes de várias colunas, assim como selecionar nomes de várias tabelas.

SQL >Comandos SQL >Distinct

A palavra-chave **SELECT** permite-nos obter todas as informações de uma coluna (ou colunas) numa tabela. Isso, claro, significa obrigatoriamente que existirão redundâncias. E se apenas pretendermos selecionar cada elemento **DISTINCT**? Isso é facilmente concretizável na linguagem SQL. Basta adicionar **DISTINCT** após **SELECT**. A sintaxe será a seguinte:

```
SELECT DISTINCT "nome_coluna"
FROM "nome_tabela";
```

Por exemplo, para selecionar todos os armazenamentos distintos na tabela ***Store_Information***,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

introduzimos

```
SELECT DISTINCT Store_Name FROM Store_Information;
```

Resultado:

Store_Name
Los Angeles
San Diego
Boston

SQL >Comandos SQL >Where

Em seguida, podemos selecionar condicionalmente os dados da tabela. Por exemplo, podemos pretender obter lojas com vendas superiores a 1.000 €. Para tal, utilizamos a palavra-chave **WHERE**. A sintaxe será a seguinte:

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "condição";
```

Por exemplo, para selecionar todas as lojas com valores superiores a 1.000 € na Tabela ***Store_Information***,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

introduzimos

```
SELECT Store_Name
FROM Store_Information
WHERE Sales > 1000;
```

Resultado:

Store_Name
Los Angeles

SQL >Comandos SQL >And Or

Na secção anterior, vimos que a palavra-chave **WHERE** pode ser utilizada para selecionar condicionalmente dados de uma tabela. Esta condição pode ser uma condição simples (como a apresentada na secção anterior) ou pode ser uma condição composta. As condições compostas são constituídas por várias condições simples ligadas através de **AND** ou **OR**. Não existe qualquer limite relativamente ao número de condições simples que podem estar presentes numa única instrução SQL.

A sintaxe de uma condição composta é a seguinte:

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "condição simples"
{[AND|OR] "condição simples"}+;
```

Os símbolos {}+ significam que a expressão entre parênteses irá ocorrer uma ou mais vezes. Note que **AND** e **OR** podem ser utilizados alternadamente. Além disso, podemos utilizar os sinais de parênteses ( ) para indicar a ordem da condição.

Por exemplo, podemos pretender selecionar todas as lojas com vendas superiores a 1 000 € ou todas as lojas com vendas inferiores a 500 € mas superiores a 275 € na Tabela ***Store_Information***,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

introduzimos

```
SELECT Store_Name
FROM Store_Information
WHERE Sales > 1000
OR (Sales < 500 AND Sales > 275);
```

Resultado:

Store_Name
Los Angeles
San Francisco

SQL >Comandos SQL >In

Na linguagem SQL, a palavra-chave **IN** pode ser utilizada de duas formas e esta secção introduz uma que está relacionada com a cláusula **WHERE**. Quando utilizada neste contexto, sabemos exatamente o valor dos valores devolvidos que pretendemos visualizar em pelo menos uma das colunas. A sintaxe para utilizar a palavra-chave **IN** é a seguinte:

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "nome_coluna" IN ('valor1', 'valor2', ...);
```

O número de valores entre parênteses pode ser um ou mais, com cada valor separado por uma vírgula. Os valores podem ser numéricos ou caracteres. Se existir apenas um valor entre parênteses, o comando é equivalente a

```
WHERE "nome_coluna" = 'valor1'
```

Por exemplo, podemos pretender selecionar todos os registos das lojas de Los Angeles e San Diego na Tabela ***Store_Information***,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

introduzimos

```
SELECT *
FROM Store_Information
WHERE Store_Name IN ('Los Angeles', 'San Diego');
```

Resultado:

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999

SQL >Comandos SQL >Between

Enquanto a palavra-chave **IN** ajuda as pessoas a limitar o critério de seleção a um ou mais valores discretos, a palavra-chave **BETWEEN** permite a seleção de um intervalo. A sintaxe da cláusula **BETWEEN** é a seguinte:

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "nome_coluna" BETWEEN "valor1' AND "valor2';
```

Este comando irá selecionar todas as linhas cuja coluna tiver um valor entre o 'valor1' e o 'valor2'.

Por exemplo, podemos querer visualizar todas as vendas entre 6 de Janeiro de 1999 e 10 de Janeiro de 2010 na Tabela ***Store_Information***,

Tabel ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

introduzimos

```
SELECT *
FROM Store_Information
WHERE Txn_Date BETWEEN '06-Jan-1999' AND '10-Jan-1999';
```

Note que a data pode estar armazenada em diferentes formatos em bases de dados diferentes. Este tutorial simplesmente seleciona um dos formatos.

Resultado:

Store_Name	Sales	Txn_Date
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

SQL >Comandos SQL >Like

LIKE é outra palavra-chave utilizada na cláusula **WHERE**. Basicamente, **LIKE** permite-lhe efetuar uma procura com base num padrão em vez de especificar exatamente o pretendido (como em **IN**) ou excluir um intervalo (como em **BETWEEN**). A sintaxe será a seguinte:

```
SELECT "nome_coluna"
FROM "nome_tabela"
WHERE "nome_coluna" LIKE {PATTERN};
```

{PATTERN} normalmente é composto por caracteres universais. A seguir são apresentados alguns exemplos:

- 'A_Z': Todas as cadeias que começam por 'A', outro caráter, e terminam em 'Z'. Por exemplo, 'ABZ' e 'A2Z' iriam ambas satisfazer a condição, enquanto 'AKKZ' não iria (porque existem dois caracteres entre A e Z em vez de um).

- 'ABC%': Todas as cadeias que começam por 'ABC'. Por exemplo, 'ABCD' e 'ABCABC' iriam satisfazer a condição.

- '%XYZ': Todas as cadeias que terminam em 'XYZ'. Por exemplo, 'WXYZ' e 'ZZXYZ' iriam satisfazer a condição.

- '%AN%': Todas as cadeias que contêm o padrão 'AN' em qualquer local. Por exemplo, 'LOS ANGELES' e 'SAN FRANCISCO' iriam satisfazer a condição.

Suponhamos que temos a seguinte tabela:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
LOS ANGELES	1500	05-Jan-1999
SAN DIEGO	250	07-Jan-1999
SAN FRANCISCO	300	08-Jan-1999
BOSTON	700	08-Jan-1999

Queremos encontrar todas as lojas cujo nome contém 'AN'. Para tal, introduzimos

```
SELECT *
FROM Store_Information
WHERE Store_Name LIKE '%AN%';
```

Resultado:

Store_Name	Sales	Txn_Date
LOS ANGELES	1500	05-Jan-1999
SAN DIEGO	250	07-Jan-1999
SAN FRANCISCO	300	08-Jan-1999

SQL >Comandos SQL >Order By

Até agora, vimos como obter dados de uma tabela utilizando os comandos **SELECT** e **WHERE**. Contudo, frequentemente precisamos de listar os resultados por uma ordem em particular. Pode ser por ordem ascendente, descendente, ou com base no valor numérico ou valor de texto. Nesses casos, podemos utilizar a palavra-chave **ORDER BY** para alcançar o objetivo.

A sintaxe para uma instrução **ORDER BY** é a seguinte:

```
SELECT "nome_coluna"
FROM "nome_tabela"
[WHERE "condição"]
ORDER BY "nome_coluna" [ASC, DESC];
```

Os símbolos [] significam que a instrução **WHERE** é opcional. Contudo, se existir uma cláusula **WHERE**, esta precede a cláusula **ORDER BY**. **ASC** significa que os resultados serão apresentados por ordem ascendente e **DESC** significa que os resultados serão apresentados por ordem descendente. Se não for especificada qualquer uma, o padrão é **ASC**.

É possível ordenar por mais do que uma coluna. Nesse caso, a cláusula **ORDER BY** acima torna-se

```
ORDER BY "nome_coluna1" [ASC, DESC], "nome_coluna2" [ASC, DESC]
```

Assumindo que selecionamos a ordem ascendente para ambas as colunas, o resultado será ordenado pela ordem ascendente de acordo com a coluna 1. Se existir uma relação para o valor da coluna 1, ordenamos por ordem ascendente a coluna 2.

Por exemplo, podemos pretender listar o conteúdo da Tabela ***Store_Information*** por montante em dólares, por ordem descendente:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

introduzimos

```
SELECT Store_Name, Sales, Txn_Date
FROM Store_Information
ORDER BY Sales DESC;
```

Resultado:

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
Boston	700	08-Jan-1999
San Francisco	300	08-Jan-1999
San Diego	250	07-Jan-1999

Para além do nome da coluna, também podemos utilizar a posição da coluna (com base na consulta SQL) para indicar a que coluna pretendemos aplicar a cláusula **ORDER BY**. A primeira coluna é 1, a segunda coluna é 2, etc. No exemplo acima, iremos obter os mesmos resultados através do seguinte comando:

```
SELECT Store_Name, Sales, Txn_Date
FROM Store_Information
ORDER BY 2 DESC;
```

SQL >Comandos SQL >Funções

Como começamos por lidar com números, a próxima questão natural é perguntar se é possível efetuar cálculos com esses números, tais como somá-los ou calcular a sua média. A resposta é sim! A linguagem SQL tem várias funções aritméticas, sendo:

- **AVG**
- **COUNT**
- **MAX**
- **MIN**
- **SUM**

A sintaxe para utilizar funções é:

```
SELECT "tipo de função" ("nome_coluna")
FROM "nome_tabela";
```

Por exemplo, se quisermos obter a soma de todas as vendas da seguinte tabela,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

introduziríamos

```
SELECT SUM (Sales) FROM Store_Information;
```

Resultado:

SUM (Sales)
2750

2750 representa a soma de todas as entradas Sales (Vendas): 1500 + 250 + 300 + 700.

Para além de utilizar funções, também é possível utilizar a linguagem SQL para efetuar tarefas simples como a adição (+) e a subtração (-). Para dados de caracteres, também se encontram disponíveis várias funções de cadeias, tais como as funções de concatenação, corte e subtração. Os diferentes vendedores de sistemas RDBMS têm diferentes implementações de cadeias e é melhor consultar as referências relativas ao seu sistema RDBMS para descobrir de que forma estas funções são utilizadas.

SQL >Comandos SQL >Count

Outra função aritmética é **COUNT**. Esta permite **COUNT** (contar) o número de linhas numa determinada tabela. A sintaxe é

```
SELECT COUNT("nome_coluna")
FROM "nome_tabela";
```

Por exemplo, se quisermos descobrir o número de entradas de uma loja na nossa tabela,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

introduziríamos

```
SELECT COUNT (Store_Name)
FROM Store_Information;
```

Resultado:

COUNT (Store_Name)
4

COUNT e **DISTINCT** podem ser utilizadas em conjunto numa instrução para obter o número de entradas diferentes numa tabela. Por exemplo, se quisermos descobrir o número de diferentes lojas, escreveríamos

```
SELECT COUNT (DISTINCT Store_Name)
FROM Store_Information;
```

Resultado:

COUNT (DISTINCT Store_Name)
3

SQL >Comandos SQL >Group By

Agora regressamos às funções agregadas. Lembra-se que utilizamos a palavra-chave **SUM** para calcular o total de vendas de todas as lojas? E se quisermos calcular o total de vendas de *cada* loja? Bem, necessitamos de duas coisas: Primeiro, é necessário certificarmo-nos que selecionamos o nome da loja, assim como o total de vendas. Segundo, é necessário garantir que todos os valores de vendas são **GROUP BY** (agrupados por) lojas. A sintaxe SQL correspondente é,

```
SELECT "nome_coluna1", SUM("nome_coluna2")
FROM "nome_tabela"
GROUP BY "nome_coluna1";
```

Vamos ilustrar através da seguinte tabela,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Queremos descobrir o total de vendas de cada loja. Para tal, introduziríamos

```
SELECT Store_Name, SUM (Sales)
FROM Store_Information
GROUP BY Store_Name;
```

Resultado:

Store_Name	SUM (Sales)
Los Angeles	1800

San Diego	250
Boston	700

A palavra-chave **GROUP BY** é utilizada ao selecionar várias colunas a partir de uma tabela (ou tabelas) e aparece pelo menos um operador aritmético na instrução **SELECT**. Quando isso acontece, é necessário **GROUP BY** (agrupar por) todas as outras colunas selecionadas, *ou seja*, todas as colunas exceto a(s) operada(s) pelo operador aritmético.

SQL >Comandos SQL >Having

Outra coisa que os utilizadores poderão pretender efetuar é limitar os resultados com base na respectiva soma (ou quaisquer outras funções agregadas). Por exemplo, podemos pretender visualizar apenas as lojas com vendas superiores a 1 500 €. Em vez de utilizarmos a cláusula **WHERE** na instrução SQL, é necessário utilizar a cláusula **HAVING**, que se encontra reservada para as funções agregadas. Normalmente a instrução **HAVING** é colocada perto do final da instrução SQL e uma instrução SQL com a cláusula **HAVING** pode ou não incluir a cláusula **GROUP BY**. A sintaxe para **HAVING** é

```
SELECT "nome_coluna1", SUM("nome_coluna2")
FROM "nome_tabela"
GROUP BY "nome_coluna1"
HAVING (condição da função aritmética);
```

Nota: a cláusula **GROUP BY** é opcional.

No nosso exemplo, a tabela ***Store_Information***,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

introduziríamos

```
SELECT Store_Name, SUM(Sales)
FROM Store_Information
GROUP BY Store_Name
HAVING SUM (Sales) > 1500;
```

Resultado:

Store_Name	SUM(Sales)
Los Angeles	1800

SQL >Comandos SQL >Alias

A seguir focamos a utilização de alias. Existem dois tipos de alias que são utilizados mais frequentemente: alias de colunas e alias de tabelas.

Resumindo, os alias de colunas existem para ajudar a organizar o resultado. No exemplo anterior, sempre que visualizamos o total de vendas, são listados como SUM(Sales). Embora sejam compreensíveis, podemos prever casos em que o cabeçalho da coluna é complicado (em especial se envolver várias operações aritméticas). A utilização de um alias de colunas tornaria os resultados muito mais compreensíveis.

O segundo tipo de alias é o alias de tabelas. Tal é obtido colocando um alias diretamente após o nome da tabela na cláusula **FROM**. Isso é conveniente quando quiser obter informações a partir de duas tabelas separadas (o termo técnico é 'efetuar uniões'). A vantagem de utilizar um alias de tabelas é imediatamente aparente visível quando falamos em uniões

Contudo, antes de abordarmos as uniões, observemos a sintaxe dos alias de colunas e tabelas:

```
SELECT "alias_tabela"."nome_coluna1" "aliar_coluna"
FROM "nome_tabela" "alias_tabela";
```

Resumidamente, ambos os tipos de alias são colocados diretamente após o item de alias, separados por um espaço em branco. Utilizamos novamente a nossa tabela ***Store_Information***,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Utilizamos os mesmos exemplos da secção **GROUP BY** de linguagem SQL, à exceção que colocamos alias de colunas e alias de tabelas:

```
SELECT A1.Store_Name Store, SUM(A1.Sales) 'Total Sales'
FROM Store_Information A1
GROUP BY A1.Store_Name;
```

Resultado:

Store	Total Sales
Los Angeles	1800
San Diego	250
Boston	700

Repare na diferença no resultado: os títulos da coluna agora são diferentes. É o resultado da utilização de utilizar alias de colunas. Repare que em vez do algo obscuro "Sum(Sales)", agora temos "Total Sales", que é muito mais compreensível como cabeçalho de uma coluna. A vantagem de utilizar um alias de tabelas não é aparente neste exemplo. Contudo, irá tornar-se evidente na **próxima secção**.

SQL >Comandos SQL >Join

Agora iremos abordar as uniões. Para efetuar uniões corretas na linguagem SQL, são necessários muitos dos elementos introduzidos até agora. Suponhamos que temos as duas tabelas seguintes:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

e queremos descobrir quais as vendas por região. Constatamos que a tabela ***Geography*** inclui informações sobre regiões e lojas e a tabela ***Store_Information*** contém informações de vendas para cada loja. Para obter as informações de vendas por região, é necessário combinar as informações das duas tabelas. Examinando as duas tabelas, descobrimos que estão ligadas através do campo comum, store_name. Primeiro iremos apresentar a instrução SQL e explicar posteriormente a utilização de cada segmento:

```
SELECT A1.Region_Name REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.Store_Name = A2.Store_Name
GROUP BY A1.Region_Name;
```

Resultado:

REGION	SALES
East	700
West	2050

As duas primeiras linhas dizem ao sistema SQL para selecionar dois campos, o primeiro é o campo "region_name" da tabela ***Geography*** (com alias como REGION), e o segundo é o somatório do campo "Sales" da tabela ***Store_Information*** (com alias como SALES). Repare que agora os alias das tabelas são utilizados aqui: *Geography* possui alias como A1, *Store_Information* com alias como A2. Sem alias, a primeira linha seria

```
SELECT Geography.Region_Name REGION, SUM(Store_Information.Sales) SALES
```

que é muito mais inadequada. Na sua essência, os alias de tabelas tornam toda a instrução SQL muito mais fácil de compreender, em especial quando se encontram incluídas várias tabelas.

A seguir, damos atenção à linha 3, a instrução **WHERE**. É aqui que especificamos a condição de união. Neste caso, queremos garantir que o conteúdo de "store_name" na tabela Geografia corresponde ao da tabela ***Store_Information*** e a forma de o fazer é defini-las de forma igual. A instrução **WHERE** é essencial para garantir que obtém o resultado correto. Sem a instrução **WHERE** correta, será obtida uma União Cartesiana. As uniões cartesianas irão produzir uma consulta com todas as combinações possíveis das duas (ou qualquer que seja o número de tabelas na instrução **FROM**). Nesse caso, uma união cartesiana iria resultar num resultado com o total de 4 x 4 = 16 linhas.

SQL >Comandos SQL >Outer Join

Anteriormente, abordamos a união esquerda, ou união interna, onde selecionamos linhas comuns nas tabelas participantes para uma união. E nos casos em que pretendemos selecionar elementos numa tabela independentemente de estarem presentes numa segunda tabela? Agora será necessário utilizar o comando **SQL OUTER JOIN**.

A sintaxe para efetuar uma união externa na linguagem SQL depende da base de dados. Por exemplo, em Oracle, iremos colocar o sinal "(+)" na cláusula **WHERE** no outro lado da tabela para a qual queremos incluir todas as linhas.

Suponhamos que temos as duas tabelas seguintes:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

e queremos descobrir o montante de vendas de todas as lojas. Se efetuarmos uma união normal, não seremos capazes de obter o que pretendemos porque iremos ignorar "New York," pois não aparece na tabela ***Store_Information***. Como tal, é necessário efetuar uma união externa nas duas tabelas acima:

```
SELECT A1.Store_Name, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.Store_Name = A2.Store_Name (+)
GROUP BY A1.Store_Name;
```

Note que neste caso estamos a utilizar a sintaxe Oracle para a união externa.

Resultado:

Store_Name	SALES
Boston	700
New York	
Los Angeles	1800
San Diego	250

Nota: NULL é obtido quando não existir qualquer correspondência na segunda tabela. Nesse caso, "New York" não aparece na tabela ***Store_Information***, e por isso a coluna "SALES" correspondente é NULL.

SQL >Comandos SQL >Função Concatenate

Por vezes é necessário combinar (concatenar) os resultados de vários campos diferentes. Cada base de dados fornece um modo de o fazer:

- MySQL: CONCAT( )
- Oracle: CONCAT( ), ||
- SQL Server: +

A sintaxe para CONCAT( ) é a seguinte:

```
CONCAT (str1, str2, str3, ...)
```

Concatenar str1, str2, str3 e quaisquer outras cadeias juntas. Note que a função Oracle CONCAT( ) apenas permite dois argumentos -- apenas duas cadeias podem ser utilizadas em simultâneo aquando da utilização desta função. Contudo, é possível concatenar mais de duas cadeias em simultâneo no Oracle utilizando '||'.

Observemos alguns exemplos. Assuma que temos a seguinte tabela:

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

Exemplo 1

MySQL/Oracle:

```
SELECT CONCAT (Region_Name, Store_Name) FROM Geography
WHERE Store_Name = 'Boston';
```

Resultado:

```
'EastBoston'
```

Exemplo 2

Oracle:

```
SELECT Region_Name || ' ' || Store_Name FROM Geography
WHERE Store_Name = 'Boston';
```

Resultado:

```
'East Boston'
```

Exemplo 3

SQL Server:

```
SELECT Region_Name + ' ' + Store_Name FROM Geography
WHERE Store_Name = 'Boston';
```

Resultado:

```
'East Boston'
```

SQL >Comandos SQL >Função Substring

A função SUBSTRING na linguagem SQL é utilizada para obter uma parte dos dados armazenados. Esta função possui nomes diferentes nas diversas bases de dados:

- MySQL: SUBSTR( ), SUBSTRING( )
- Oracle: SUBSTR( )
- SQL Server: SUBSTRING( )

A seguir são apresentadas as utilizações mais frequentes (iremos utilizar SUBSTR( ) aqui):

SUBSTR (str, pos)

Selecionar todos os caracteres de <str> a começar pela posição <pos>. Note que esta sintaxe não é suportada pelo SQL Server.

SUBSTR (str, pos, len)

A começar pelo <pos> caráter na cadeia <str> e selecionar os caracteres <len> seguintes.

Assuma que temos a seguinte tabela:

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

Exemplo 1

```
SELECT SUBSTR (Store_Name, 3)
FROM Geography
WHERE Store_Name = 'Los Angeles';
```

Resultado:

```
's Angeles'
```

Exemplo 2

```
SELECT SUBSTR (Store_Name, 2, 4)
FROM Geography
WHERE Store_Name = 'San Diego';
```

Resultado:

```
'an D'
```

SQL >Comandos SQL >Função Trim

A função TRIM em linguagem SQL é utilizada para remover prefixos ou sufixos especificados de uma cadeia. O padrão mais comum a ser removido dão os espaços em branco. Esta função possui nomes diferentes nas diversas bases de dados:

- MySQL: TRIM( ), RTRIM( ), LTRIM( )
- Oracle: RTRIM( ), LTRIM( )
- SQL Server: RTRIM( ), LTRIM( )

A sintaxe para estas funções de corte é:

TRIM ([[LOCATION] [remstr] FROM] str): [LOCATION] pode ser LEADING, TRAILING ou BOTH. Esta função elimina o padrão [remstr] do início de uma cadeia, do final de uma cadeia ou ambos. Se não for especificado [remstr], os espaços em branco são removidos.

LTRIM (str): Remove todos os espaços em branco do início da cadeia.

RTRIM (str): Remove todos os espaços em branco do final da cadeia.

Exemplo 1

```
SELECT TRIM('   Sample   ');
```

Resultado:

```
'Sample'
```

Exemplo 2

```
SELECT LTRIM('   Sample   ');
```

Resultado:

```
'Sample   '
```

Exemplo3

```
SELECT RTRIM('   Sample   ');
```

Resultado:

```
'   Sample'
```

SQL > Comandos SQL > Função Length

A função LENGTH em SQL é utilizada para obter o comprimento de uma cadeia. Esta função possui nomes diferentes nas diversas bases de dados:

- MySQL: LENGTH ( )
- Oracle: LENGTH ( )
- SQL Server: LEN ( )

A sintaxe da função LENGTH é a seguinte:

Length (str): Encontrar o comprimento da cadeia *str*.

Observemos alguns exemplos. Assuma que temos a seguinte tabela:

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

Exemplo 1

```
SELECT Length (Store_Name)
FROM Geography
WHERE Store_Name = 'Los Angeles';
```

Resultado:

```
11
```

Exemplo 1

```
SELECT Region_Name, Length (Region_Name)
FROM Geography;
```

Resultado:

Region_Name	Length (Region_Name)
East	4
East	4
West	4
West	4

SQL > **Comandos SQL** > **Função Replace**

A função **Replace** em SQL é utilizada para atualizar o conteúdo de uma cadeia. A chamada de função é **REPLACE( )** para MySQL, Oracle e SQL Server. A sintaxe da função **REPLACE** é:

```
Replace (str1, str2, str3)
```

Em str1, encontrar onde str2 ocorre e substituir por str3.

Assuma que temos a seguinte tabela:

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

Se aplicarmos a seguinte função **REPLACE**:

```
SELECT REPLACE (Region_Name, 'ast', 'astern')
FROM Geography;
```

Resultado:

Region_Name
Eastern
Eastern
West
West

SQL > **Comandos SQL** > **Função Dateadd**

A função **DATEADD** é utilizada para adicionar um intervalo a uma data. Esta função encontra-se disponível no SQL Server.

A utilização da função **DATEADD** é

```
DATEADD (datepart, number, expression)
```

em que o tipo de dados de <expression> é algum tipo de data, hora ou data e hora. <number> é um número inteiro (pode ser positivo ou negativo). <datepart> pode ser um dos seguintes elementos:

datepart	Abreviatura
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
hour	hh
minute	mi, n
second	ss, s
millisecond	ms

microsecond	mcs
nanosecond	ns
TZoffset	tz
ISO_WEEK	isowk, isoww

O resultado obtido tem o mesmo tipo de dados de <expression>.

Exemplo: A instrução SQL

```
SELECT DATEADD (day, 10, '2000-01-05 00:05:00.000');
```

produz o seguinte resultado:

```
'2000-01-15 00:05:00.000'
```

SQL > **Comandos SQL** > **Função Datediff**

A função **DATEDIFF** é utilizada para calcular a diferença entre dois dias e é utilizada no sistema MySQL e SQL Server. A sintaxe desta função de data é diferente entre estas duas bases de dados, pelo que cada uma delas é abordada abaixo:

MySQL:

A utilização da função **DATEDIFF** no sistema MySQL é

```
DATEDIFF (expression1, expression2)
```

em que o tipo de dados de <expression1> e <expression2> é DATE ou DATETIME. O resultado é <expression1> - <expression2>.

Exemplo: A instrução SQL

```
SELECT DATEDIFF('2000-01-10', '2000-01-05');
```

produz o seguinte resultado:

```
5
```

Isso deve-se ao fato de 2000-01-10 ser 5 dias após 2000-01-05.

SQL Server:

A utilização da função DATEDIFF no sistema SQL Server é

```
DATEDIFF (datepart, expression1, expression2)
```

em que o tipo de dados de <expression1> e <expression2> > é um tipo de data, hora ou data e hora. O resultado é <expression2> - <expression1>. datepart pode ser um dos seguintes:

datepart	Abreviatura

year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns
TZoffset	tz
ISO_WEEK	isowk, isoww

Exemplo: A instrução SQL

```
SELECT DATEDIFF(day, '2000-01-10', '2000-01-05');
```

produz o seguinte resultado:

```
-5
```

Isso deve-se ao fato de 2000-01-05 ser 5 dias antes de 2000-01-10.

SQL > **Comandos SQL** > **Função Datepart**

DATEPART é uma função do SQL Server que extrai uma parte específica do valor de data/hora. A sua sintaxe será a seguinte:

```
DATEPART (part_of_day, expression)
```

em que part_of_day pode ser composto pelo seguinte:

datepart	Afkorting
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
hour	hh
minute	mi, n

second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns
TZoffset	tz
ISO_WEEK	isowk, isoww

Exemplo 1

```
SELECT DATEPART (yyyy, '2000-01-20');
```

Resultado:

```
2000
```

Exemplo 2

```
SELECT DATEPART(dy, '2000-02-10');
```

Resultado:

```
41
```

2000-02-10 é o 41.º dia do ano de 2000.

SQL > Comandos SQL > Função Getdate

A função **GETDATE** é utilizada para obter a hora do sistema atual da base de dados no SQL Server. A sua sintaxe é

```
GETDATE ( )
```

GETDATE não requer qualquer argumento.

Exemplo: A instrução SQL

```
SELECT GETDATE ( );
```

produz o seguinte resultado:

```
'2000-03-15 00:05:02.123'
```

A função **GETDATE** é muito útil quando necessitamos de registar a hora em que uma transação em particular ocorre. No SQL Server, basta introduzirmos o valor da função **GETDATE( )** na tabela para obter o resultado. Também podemos definir o valor padrão de uma coluna como sendo **GETDATE( )** para alcançar o mesmo objetivo.

O equivalente a **GETDATE** para os sistemas Oracle e MySQL é **SYSDATE**.

SQL > Comandos SQL > Função Sysdate

A função **SYSDATE** é utilizada para obter a hora do sistema atual da base de dados nos sistemas Oracle e MySQL.

Oracle:

A sintaxe de **SYSDATE** em Oracle é simplesmente

```
SYSDATE
```

Não requer qualquer argumento.

Exemplo: A instrução SQL

```
SELECT SYSDATE FROM DUAL;
```

produz o seguinte resultado:

```
'16-JAN-2000'
```

MySQL:

A sintaxe de **SYSDATE** em MySQL é simplesmente

```
SYSDATE ( )
```

Não requer qualquer argumento.

Exemplo: A instrução SQL

```
SELECT SYSDATE ( );
```

produz o seguinte resultado:

```
'2000-01-16 09:06:22'
```

O equivalente a **SYSDATE** para o SQL Server é **GETDATE**.

SQL >Manipulação de Tabelas >Create Table

As tabelas são a estrutura básica em que os dados são armazenados na base de dados. Dado que na maior parte dos casos não existe uma forma de o vendedor da base de dados saber antecipadamente quais são as suas necessidades em termos de armazenamento de dados, é provável que seja necessário criar tabelas na base de dados. Muitas ferramentas de bases de dados permitem-lhe criar tabelas sem programar em linguagem SQL, mas como as tabelas são o receptáculo de todos os dados, é importante incluir a sintaxe **CREATE TABLE** neste tutorial.

Antes de abordarmos a sintaxe SQL para **CREATE TABLE**, é boa ideia compreender a estrutura de uma tabela. As tabelas encontram-se divididas em linhas e colunas. Cada linha representa um dado individual e cada coluna pode ser vista como uma representação dessa peça de dados. Assim, por exemplo, se tivermos uma tabela para registar as informações dos clientes, as colunas podem incluir informações como Primeiro Nome, Apelido, Morada, Cidade, País, Data de Nascimento, etc. Como tal, ao especificarmos uma tabela, incluímos os cabeçalhos das colunas e os tipos de dados para essa coluna em particular.

Assim sendo, quais são os tipos de dados? Normalmente, os dados encontram-se disponíveis numa variedade de formas. Podem ser um número inteiro (tal como 1), um número real (tal como 0,55), uma cadeia (tal como 'sql'), uma expressão de data/hora (tal como '2000-JAN-25 03:22:22') ou até mesmo um formato binário. Ao especificarmos uma tabela, é necessário especificarmos o tipo de dados associados a cada coluna(ou seja, iremos especificar o 'Primeiro Nome' do tipo de char(50) - o que significa uma cadeia com 50 caracteres). Um aspecto a destacar é que diferentes bases de dados relacionais permitem diferentes tipos de dados, pelo que é aconselhável consultar primeiro um documento de referência relativo à base de dados específica.

A sintaxe SQL para **CREATE TABLE** é

```
CREATE TABLE "nome_tabela"
("coluna 1" "tipo_dados_para_coluna_1",
"coluna 2" "tipo_dados_para_coluna_2",
... );
```

Assim, se pretendermos criar uma tabela de clientes conforme acima especificado, introduziríamos

```
CREATE TABLE Customer
(First_Name char(50),
Last_Name char(50),
Address char(50),
City char(50),
Country char(25),
Birth_Date datetime);
```

Por vezes, pretendemos fornecer um valor padrão para cada coluna. É utilizado um valor padrão quando não especificar um valor para a coluna ao introduzir dados na tabela. Para especificar um valor padrão, adicione "Default [valor]" após a declaração do tipo de dados. No exemplo acima, que quiser que o padrão da coluna "Address" passe para "Unknown" e "City" para "Rio de Janeiro", escreveria

```
CREATE TABLE Customer
(First_Name char(50),
Last_Name char(50),
Address char(50) default 'Unknown',
City char(50) default 'Rio de Janeiro',
Country char(25),
Birth_Date datetime);
```

Também pode limitar o tipo de informações que uma tabela/coluna pode suportar. Esse passo é efetuado através da palavra-chave **CONSTRAINT**, abordada a seguir.

SQL >Manipulação de Tabelas >Constraint

Pode colocar restrições para limitar o tipo de dados a introduzir numa tabela. Essas restrições podem ser especificadas quando a tabela for primeiro criada através da instrução **CREATE TABLE** ou após a tabela já ter sido criada através da instrução **ALTER TABLE**.

Alguns tipos comuns de restrições incluem o seguinte:

- **NOT NULL Constraint**: Garante que uma coluna não pode ter o valor NULL.
- **DEFAULT Constraint**: Fornece um valor padrão para uma coluna quando nenhum é especificado.
- **UNIQUE Constraint**: Garante que todos os valores numa coluna são diferentes.
- **CHECK Constraint**: Garante que todos os valores numa coluna satisfazem um determinado critério.
- **Primary Key Constraint**: Utilizado para identificar de forma única uma linha na tabela.
- **Foreign Key Constraint**: Utilizado para garantir a integridade referencial dos dados.

Cada restrição é abordada nas próximas secções.

SQL >Manipulação de Tabelas > NOT NULL Constraint

Por defeito, uma coluna pode suportar NULL. Se não quiser permitir o valor NULL numa coluna, deverá colocar uma restrição nessa coluna a especificar que NULL agora não é um valor permitido.

Por exemplo, na seguinte instrução,

```
CREATE TABLE Customer
(SID integer NOT NULL,
Last_Name varchar (30) NOT NULL,
First_Name varchar(30));
```

As colunas "SID" e "Last_Name" não podem incluir NULL, enquanto "First_Name" pode incluir NULL.

Uma tentativa de execução a seguinte instrução SQL,

```
INSERT INTO Customer (Last_Name, First_Name) VALUES ('Damaso','Joana');
```

irá provocar um erro porque irá fazer com que a coluna "SID" seja NULL, o que viola a restrição **NOT NULL** nessa coluna.

SQL > Manipulação de Tabelas > DEFAULT Constraint

A restrição **DEFAULT** fornece um valor padrão a uma coluna quando a instrução **INSERT INTO** não fornecer um valor específico. Por exemplo, se criarmos uma tabela conforme abaixo apresentado:

```
CREATE TABLE Student
(Student_ID integer Unique,
Last_Name varchar (30),
First_Name varchar (30),
Score DEFAULT 80);
```

e executarmos a seguinte instrução SQL,

```
INSERT INTO Student (Student_ID, Last_Name, First_Name) VALUES (10, 'Johnson', 'Rick');
```

A tabela irá ter o seguinte aspecto:

Student_ID	Last_Name	First_Name	Score
10	Johnson	Rick	80

Embora não tenhamos especificado um valor para a coluna "Score" na instrução **INSERT INTO**, não lhe é atribuído o valor padrão 80 porque já definimos 80 como o valor padrão desta coluna.

SQL > Manipulação de Tabelas > UNIQUE Constraint

A restrição **UNIQUE** garante que todos os valores numa coluna são diferentes.

Por exemplo, na seguinte instrução **CREATE TABLE**,

```
CREATE TABLE Customer
(SID integer UNIQUE,
Last_Name varchar (30),
First_Name varchar (30));
```

a coluna "SID" tem uma restrição única e, por conseguinte, não pode incluir valores duplicados. Tal restrição não suporta as colunas "Last_Name" e "First_Name". Assim sendo, se a tabela já possuir as seguintes linhas:

SID	Last_Name	First_Name
1	Pessoa	Stella
2	Villa	Bruno
3	Feitor	Sara

Executar a seguinte instrução SQL,

```
INSERT INTO Customer VALUES ( 3, 'Cabral', 'Diana' );
```

irá provocar um erro, pois '3' já existe na coluna SID e, como tal, tentar introduzir outra linha com esse valor viola a restrição **UNIQUE**.

Note que uma coluna que seja especificada como chave primária também deve ser única. Ao mesmo tempo, uma coluna que seja única pode ou não pode ser uma chave primária. Além disso, é possível definir várias restrições **UNIQUE** numa tabela.

SQL > **Manipulação de Tabelas** > **CHECK Constraint**

A restrição **CHECK** garante que todos os valores numa coluna satisfazem determinadas condições. Uma vez definida, a base de dados apenas irá introduzir uma nova linha ou atualizar uma existente se o novo valor satisfizer a restrição **CHECK**. A restrição **CHECK** é utilizada para garantir a qualidade dos dados.

Por exemplo, na seguinte instrução **CREATE TABLE**,

```
CREATE TABLE Customer
(SID integer CHECK (SID > 0),
Last_Name varchar (30),
First_Name varchar (30));
```

A coluna "SID" tem uma restrição -- o seu valor apenas deve incluir números inteiros superiores a 0. Como tal, tentar executar a seguinte instrução

```
INSERT INTO Customer VALUES (-3, 'Brito', 'Rita' );
```

irá dar origem a um erro, pois os valores de SID devem ser superiores a 0.

Note que desta vez a restrição **CHECK** não é importa por linguagem MySQL.

SQL >**Manipulação de Tabelas** >**Chave Primária**

Uma chave primária é utilizada da identificar de forma única cada linha numa tabela. Pode fazer parte do próprio registo atual ou pode ser um campo artificial (um que não tenha nada que ver com o registo atual). Uma chave primária pode ser composta por um ou mais campos numa tabela. Quando são utilizados vários campos como chave primária, são denominados por chave composta.

As chaves primárias podem ser especificadas quando a tabela é criada (utilizando **CREATE TABLE**) ou alterando a estrutura da tabela existente (utilizando **ALTER TABLE**).

Abaixo são apresentados exemplos para a especificação de uma chave primária ao criar uma tabela:

MySQL:

```
CREATE TABLE Customer
(SID integer,
Last_Name varchar(30),
First_Name varchar(30),
PRIMARY KEY (SID) );
```

Oracle:

```
CREATE TABLE Customer
(SID integer PRIMARY KEY,
Last_Name varchar(30),
First_Name varchar(30) );
```

SQL Server:

```
CREATE TABLE Customer
(SID integer PRIMARY KEY,
Last_Name varchar(30),
First_Name varchar(30) );
```

Abaixo são apresentados exemplos para a especificação de uma chave primária ao alterar uma tabela:

MySQL:

```
ALTER TABLE Customer ADD PRIMARY KEY (SID);
```

Oracle:

```
ALTER TABLE Customer ADD PRIMARY KEY (SID);
```

SQL Server:

```
ALTER TABLE Customer ADD PRIMARY KEY (SID);
```

Nota: Antes de utilizar o comando **ALTER TABLE** para adicionar uma chave primária, deve certificar-se de que o campo se encontra definido como 'NOT NULL' -- por outras palavras, NULL não pode ser um valor aceite para esse campo.

SQL >Manipulação de Tabelas >Chave Externa

Uma chave externa é um campo (ou campos) que aponta para a chave primária de outra tabela. O objetivo da chave externa é garantir a integridade referencial dos dados Por outras palavras, apenas os valores suportados que supostamente devem aparecer na base de dados são permitidos.

Por exemplo, suponhamos que temos duas tabelas, uma tabela ***CUSTOMER*** que inclui todos os dados dos clientes e uma tabela ***ORDERS*** que inclui todas as encomendas dos clientes. A restrição prende-se com o fato de todas as encomendas deverem ser associadas a um cliente que já esteja na tabela ***CUSTOMER***. Nesse caso, iremos colocar uma chave externa na tabela ***ORDERS*** e fazer com que se relacione com a chave primária da tabela ***CUSTOMER***. Deste modo, podemos garantir que todas as encomendas na tabela ***ORDERS*** estão relacionadas com um cliente na tabela ***CUSTOMER***. Por outras palavras, a tabela ***ORDERS*** não pode conter informações sobre um cliente que não se encontre na tabela ***CUSTOMER***.

A estrutura destas duas tabelas seria a seguinte:

Tabela ***CUSTOMER***

Nome da Coluna	Característica
SID	Chave Primária
Last_Name	
First_Name	

Tabela ***ORDERS***

Nome da Coluna	Característica
Order_ID	Chave Primária
Order_Date	
Customer_SID	Chave Externa
Amount	

No exemplo acima apresentado, a coluna Customer_SID na tabela ***ORDERS*** é uma chave externa a apontar para a coluna SID na tabela ***CUSTOMER***.

Abaixo são apresentados exemplos de como especificar uma chave externa ao criar a tabela ***ORDERS***:

MySQL:

```
CREATE TABLE ORDERS
(Order_ID integer,
Order_Date date,
Customer_SID integer,
Amount double,
PRIMARY KEY (Order_ID),
FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID));
```

Oracle:

```
CREATE TABLE ORDERS
(Order_ID integer PRIMARY KEY,
Order_Date date,
Customer_SID integer REFERENCES CUSTOMER (SID),
Amount double);
```

SQL Server:

```
CREATE TABLE ORDERS
(Order_ID integer PRIMARY KEY,
Order_Date datetime,
Customer_SID integer REFERENCES CUSTOMER (SID),
Amount double);
```

Abaixo são apresentados exemplos para a especificação de uma **chave externa** ao alterar uma tabela. Isso assume que a tabela ***ORDERS*** foi criada e que a chave externa ainda não foi introduzida:

MySQL:

```
ALTER TABLE ORDERS
ADD FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID);
```

Oracle:

```
ALTER TABLE ORDERS
ADD (CONSTRAINT fk_orders1) FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID);
```

SQL Server:

```
ALTER TABLE ORDERS
ADD FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID);
```

SQL >Manipulação de Tabelas >Create View

As vistas podem ser consideradas como tabelas virtuais. Regra geral, uma tabela tem um conjunto de definições e armazena fisicamente os dados. Uma vista também tem um conjunto de definições, que são criadas sobre tabela(s) ou outra(s) vista(s), e não armazena fisicamente os dados.

A sintaxe para criar uma vista é a seguinte:

```
CREATE VIEW "nome_vista" AS "Instrução SQL";
```

"Instrução SQL" pode ser qualquer uma das instruções SQL abordadas neste tutorial.

Utilizemos um exemplo simples como ilustração. Assuma que temos a seguinte tabela:

Tabela ***Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	Datetime

e que queremos criar uma vista denominada ***V_Customer*** que contenha apenas as colunas Primeiro_nome, Apelido e País desta tabela, escreveríamos

```
CREATE VIEW V_Customer
AS SELECT First_Name, Last_Name, Country
FROM Customer;
```

Agora possuímos uma vista denominada ***V_Customer*** com a seguinte estrutura:

View ***V_Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Country	char(25)

Também podemos utilizar uma vista para aplicar uniões a duas tabelas. Nesse caso, os utilizadores apenas visualizam uma em vez de duas tabelas e a instrução SQL que os utilizadores devem emitir torna-se muito mais simples. Suponhamos que temos as duas tabelas seguintes:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

e queremos criar uma vista com informações de vendas por região. Iríamos emitir a seguinte instrução SQL:

```
CREATE VIEW V_REGION_SALES
AS SELECT A1.Region_Name REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.Store_Name = A2.Store_Name
GROUP BY A1.Region_Name;
```

Assim obtemos uma vista, ***V_REGION_SALES***, que foi definida para armazenar registos de vendas por região. Se quisermos descobrir o conteúdo desta vista, escrevemos

```
SELECT * FROM V_REGION_SALES;
```

Resultado:

REGION	SALES
East	700
West	2050

SQL >Manipulação de Tabelas >Create Index

Os índices ajudam-nos a obter mais rapidamente dados das tabelas. Usemos um exemplo para ilustrar este ponto: Digamos que estamos interessados em ler sobre como cultivar pimentos num livro de jardinagem. Em vez de lermos o livro do início ao fim até encontrarmos uma secção sobre pimentos, é muito mais fácil utilizar a secção do índice no final do livro, localizar as páginas que contêm informações sobre pimentos e, em seguida, consultar diretamente essas páginas. Consultar primeiro o índice permite poupar tempo e é o método mais eficiente para localizar as informações necessárias.

O mesmo princípio é aplicável para a obtenção de dados a partir de uma tabela de base de dados. Sem um índice, o sistema da base de dados lê toda a tabela (este processo é denominado 'pesquisa da tabela') para localizar as informações pretendidas. Através de um índice adequado, o sistema da base de dados pode percorrer primeiro o índice para descobrir onde obter os dados e, em seguida, aceder diretamente aos locais para obter os dados necessários. Assim é muito mais rápido.

Como tal, muitas vezes é preferível criar índices nas tabelas. Um índice pode abranger uma ou mais colunas. A sintaxe geral para criar um índice é:

```
CREATE INDEX "nome_indice" ON "nome_tabela" (nome_coluna);
```

Suponhamos que temos a seguinte tabela:

Tabela ***Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	Datetime

e queremos criar um índice da coluna Apelido, escreveríamos

```
CREATE INDEX IDX_CUSTOMER_LAST_NAME
ON Customer (Last_Name);
```

Se quisermos criar um índice de Cidade e País, escreveríamos

```
CREATE INDEX IDX_CUSTOMER_LOCATION
ON Customer (City, Country);
```

Não existe uma regra rígida sobre qual o nome a atribuir a um índice. O método geralmente aceite é colocar um prefixo, tal como "IDX_", antes de um nome de índice de modo a evitar confusões com outros objetos da base de dados. Também é boa ideia fornecer informações sobre que tabela e coluna(s) o índice é utilizado.

Note que a sintaxe exata para **CREATE INDEX** pode ser diferente para diferentes bases de dados. Deve consultar o manual de referência da sua base de dados para obter a sintaxe precisa.

SQL >Manipulação de Tabelas >Alter Table

Assim que uma tabela for criada na base de dados, muitas vezes o utilizador poderá querer alterar a estrutura da tabela. Os casos típicos incluem o seguinte:

- Adicionar uma coluna
- Remover uma coluna
- Alterar o nome de uma coluna
- Alterar o tipo de dados de uma coluna

Note que o acima apresentado não constitui uma lista exaustiva. Existem outros momentos em que ALTER TABLE é utilizado para alterar a estrutura da tabela, tal como alterar a especificação da chave primária ou adicionar uma restrição única a uma coluna.

A sintaxe SQL para **ALTER TABLE** é:

```
ALTER TABLE "nome_tabela"
[alter specification];
```

[alter specification] depende do tipo de alteração que pretendemos efetuar. Para os fins acima citados, as instruções [alter specification] são:

- Adicionar uma coluna: ADD "coluna 1" "tipo de dados para a coluna 1"
- Remover uma coluna: DROP "coluna 1"
- Alterar o nome de uma coluna: CHANGE "antigo nome da coluna" "novo nome da coluna" "tipo de dados para novo nome da coluna"
- Alterar o tipo de dados de uma coluna: MODIFY "coluna 1" "novo tipo de dados"

Analisemos cada um dos exemplos acima apresentados utilizando a tabela "customer" na secção **CREATE TABLE**:

Tabela ***Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	Datetime

Primeiro, queremos adicionar uma coluna denominada "Gender" a esta tabela. Para tal, introduzimos:

```
ALTER TABLE Customer ADD Gender char(1);
```

Estrutura da tabela obtida:

Tabela ***Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	Datetime
Gender	char(1)

A seguir, queremos renomear "Address" para "Addr". Para tal, introduzimos:

```
ALTER TABLE Customer CHANGE Address Addr char(50);
```

Estrutura da tabela obtida:

Tabela ***Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Addr	char(50)
City	char(50)
Country	char(25)
Birth_Date	Datetime
Gender	char(1)

Em seguida, queremos alterar o tipo de dados de "Addr" para char(30). Para tal, introduzimos:

```
ALTER TABLE Customer MODIFY Addr char(30);
```

Estrutura da tabela obtida:

Tabela ***Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)
Birth_Date	Datetime
Gender	char(1)

Finalmente, queremos eliminar a coluna "Gender". Para tal, introduzimos:

```
ALTER TABLE Customer DROP Gender;
```

Estrutura da tabela obtida:

Tabela ***Customer***

Nome da Coluna	Tipo de Dados
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)

Birth_Date	Datetime

SQL >Manipulação de Tabelas >Drop Table

Por vezes podemos pretender remover uma tabela da base de dados por qualquer motivo. De fato, seria problemático se tal não fosse possível, pois poderia ser um pesadelo para o DBA (administrador da base de dados). Felizmente, a linguagem SQL permite-nos fazê-lo através do comando **DROP TABLE**. A sintaxe para **DROP TABLE** é

```
DROP TABLE "nome_tabela";
```

Assim, se quisermos eliminar a tabela denominada "customer" que criamos na secção **CREATE TABLE**, basta escrever

```
DROP TABLE Customer;
```

SQL >Manipulação de Tabelas >Truncate Table

Por vezes queremos remover todos os dados de uma tabela. Uma forma de o fazer é através do comando **DROP TABLE**, abordado na **secção anterior**. Mas e se apenas quisermos remover todos os dados , mas não a própria tabela? Para tal, podemos utilizar o comando **TRUNCATE TABLE**. A sintaxe para **TRUNCATE TABLE** é

```
TRUNCATE TABLE "nome_tabela";
```

Assim, se quisermos truncar a tabela denominada "customer" que criamos em **SQL CREATE TABLE**, basta escrever

```
TRUNCATE TABLE Customer;
```

SQL >Manipulação de Tabelas >Insert Into

Nas secções anteriores, vimos como obter informações das tabelas. Mas de que forma estas linhas de dados são introduzidas inicialmente nas tabelas? É isso que esta secção, que aborda a instrução **INSERT** e a secção seguinte, que aborda a instrução **UPDATE**, explicam.

Na linguagem SQL, existem essencialmente duas formas de **INSERT** (Inserir) dados numa tabela: Uma é inserir uma linha de cada vez, a outra é inserir várias linhas em simultâneo. Vamos abordar como podemos **INSERT** (Inserir) dados numa linha de cada vez:

A sintaxe para inserir dados numa linha da tabela de cada vez é a seguinte:

```
INSERT INTO "nome_tabela" ("coluna 1", "coluna 2", ...)
VALUES ("valor 1", "valor 2", ...);
```

Assumindo que temos uma tabela com a seguinte estrutura,

Tabela ***Store_Information***

Nome da Coluna	Tipo de Dados
Store_Name	char(50)
Sales	Float

Txn_Date	Datetime

e agora quisermos inserir uma linha adicional na tabela a representando os dados de vendas de Los Angeles no dia 10 de Janeiro de 1999. Nesse dia, esta loja efetuou vendas no valor de 900 €. Deste modo, iremos utilizar o seguinte script SQL:

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
VALUES ('Los Angeles', 900, '10-Jan-1999' );
```

O segundo tipo de instrução **INSERT INTO** permite-nos inserir várias linhas numa tabela. Ao contrário do exemplo anterior, em que inserimos uma única linha especificando os seus valores para todas as colunas, agora utilizamos uma instrução **SELECT** para especificar os dados que queremos inserir na tabela. Se estiver a pensar que isso significa utilizar informações de outra tabela, está certo. A sintaxe será a seguinte:

```
INSERT INTO "tabela 1" ("coluna 1", "coluna 2", ...)
SELECT "coluna 3", "coluna 4", ...
FROM "tabela 2";
```

Note que esta é a forma simples. A instrução completa pode facilmente conter as cláusulas **WHERE**, **GROUP BY**, e **HAVING**, assim como alias e uniões de tabelas.

Assim, por exemplo, se pretendermos uma tabela, ***Store_Information***, que recolha as informações de vendas para o ano de 1998 e já souber que os dados de origem se encontram na tabela ***Sales_Information***, iremos escrever:

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
SELECT Store_Name, Sales, Txn_Date
FROM Sales_Information
WHERE Year (Txn_Date) = 1998;
```

Aqui foi utilizada a sintaxe do sistema Servidor SQL para extrair a informação do ano de uma data. Outras bases de dados relacionadas terão uma sintaxe diferente. Por exemplo, no sistema Oracle, irá utilizar **TO_CHAR (Txn_Date, 'yyyy') = 1998**.

SQL >**Manipulação de Tabelas** >**Update**

Assim que existirem dados na tabela, podemos chegar à conclusão que é necessário modificar os dados. Par tal, podemos utilizar o comando **UPDATE**. A sintaxe para tal é

```
UPDATE "nome_tabela"
SET "coluna 1" = [novo valor]
WHERE "condição";
```

Por exemplo, se possuirmos atualmente uma tabela conforme abaixo apresentado:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

e descobrirmos que as vendas em Los Angeles no dia 08-Jan-1999 foram realmente de 500 € e não de 300 € e, como tal, essa entrada em particular deve ser atualizada. Para tal, utilizamos a seguinte instrução SQL:

```
UPDATE Store_Information
SET Sales = 500
WHERE Store_Name = 'Los Angeles'
AND Txn_Date = '08-Jan-1999';
```

A tabela obtida seria semelhante a

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	500	08-Jan-1999
Boston	700	08-Jan-1999

Neste caso, existe apenas uma linha que satisfaz a condição da cláusula **WHERE**. Se existirem várias filas que satisfaçam a condição, é necessário modificá-las todas.

Também é possível **UPDATE** (Atualizar) várias colunas em simultâneo. Nesse caso, a sintaxe seria semelhante à seguinte:

```
UPDATE "nome_tabela"
SET kolom 1 = [valor 1], kolom 2 = [valor 2]
WHERE "condição";
```

SQL >**Manipulação de Tabelas** >**Delete From**

Por vezes, podemos pretender remover registos de uma tabela. Par tal, podemos utilizar o comando **DELETE FROM**. A sintaxe para tal é

```
DELETE FROM "nome_tabela"
WHERE "condição";
```

É mais fácil utilizar um exemplo. Se possuirmos atualmente uma tabela conforme abaixo apresentado:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

e decidirmos não manter quaisquer informações sobre Los Angeles nesta tabela. Para tal, escrevemos a seguinte instrução SQL:

```
DELETE FROM Store_Information
WHERE Store_Name = 'Los Angeles';
```

Agora o conteúdo da tabela seria semelhante a

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
San Diego	250	07-Jan-1999
Boston	700	08-Jan-1999

SQL >Comandos SQL >Union

O objetivo do comando SQL **UNION** é combinar os resultados de duas consultas. A este respeito, **UNION** é algo semelhante a **JOIN**, pois ambos são utilizados para relacionar informações de várias tabelas. Uma restrição de **UNION** é que todas as colunas correspondentes devem possuir o mesmo tipo de dados. Além disso, ao utilizar **UNION**, apenas são selecionados valores diferentes (semelhante a **SELECT DISTINCT**).

A sintaxe será a seguinte:

```
[Instrução SQL 1]
UNION
[Instrução SQL 2];
```

Assuma que temos as seguintes duas tabelas:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Internet_Sales***

Txn_Date	Sales
07-Jan-1999	250
10-Jan-1999	535
11-Jan-1999	320
12-Jan-1999	750

e quisermos descobrir todas as datas em que ocorreram transações de vendas. Para tal, utilizamos a seguinte instrução SQL:

```
SELECT Txn_Date FROM Store_Information
UNION
SELECT Txn_Date FROM Internet_Sales;
```

Resultado:

Txn_Date
05-Jan-1999
07-Jan-1999
08-Jan-1999
10-Jan-1999
11-Jan-1999
12-Jan-1999

Note que se escrevermos **SELECT DISTINCT Txn_Date** para uma ou ambas as instruções SQL, iremos obter o mesmo conjunto de resultados.

SQL >Linguagem SQL avançada >Union All

O objetivo do comando SQL **UNION ALL** também é combinar os resultados de duas consultas. A diferença entre **UNION ALL** e **UNION** é que enquanto **UNION** apenas seleciona valores diferentes, **UNION ALL** seleciona todos os valores.

De syntaxis voor **UNION ALL** is als volgt:

```
[Instrução SQL 1]
UNION ALL
[Instrução SQL 2];
```

Vamos utilizar um exemplo igual ao da secção anterior para ilustrar a diferença. Suponhamos que temos as duas tabelas seguintes:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Internet_Sales***

Txn_Date	Sales
07-Jan-1999	250
10-Jan-1999	535
11-Jan-1999	320
12-Jan-1999	750

e quisermos descobrir todas as datas em que ocorreram transações de vendas na loja, assim como as datas em que ocorreram vendas através da Internet. Para tal, utilizamos a seguinte instrução SQL:

```
SELECT Txn_Date FROM Store_Information
UNION ALL
SELECT Txn_Date FROM Internet_Sales;
```

Resultado:

Txn_Date
05-Jan-1999
07-Jan-1999
08-Jan-1999
08-Jan-1999
07-Jan-1999
10-Jan-1999
11-Jan-1999
12-Jan-1999

SQL >Linguagem SQL avançada >Intersect

Semelhante ao comando **UNION**, **INTERSECT** também funciona com duas instruções SQL. A diferença é que enquanto **UNION** UNION essencialmente funciona como um operador **OR** (o valor é selecionado se aparecer na primeira ou na segunda instrução), o comando **INTERSECT** funciona como um operador **AND** (o valor apenas é selecionado se aparecer em ambas as instruções).

A sintaxe será a seguinte:

```
[Instrução SQL 1]
INTERSECT
[Instrução SQL 2];
```

Suponhamos que temos as duas tabelas seguintes:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Internet_Sales***

Txn_Date	Sales
07-Jan-1999	250
10-Jan-1999	535
11-Jan-1999	320
12-Jan-1999	750

e queremos descobrir todas as datas em que ocorreram vendas na loja e através da Internet. Para tal, utilizamos a seguinte instrução SQL:

```
SELECT Txn_Date FROM Store_Information
INTERSECT
SELECT Txn_Date FROM Internet_Sales;
```

Resultado:

Txn_Date
07-Jan-1999

Note que o comando **INTERSECT** apenas irá apresentar valores diferentes.

SQL >Linguagem SQL avançada >Minus

O comando **MINUS** funciona com duas instruções SQL. Recolhe todos os resultados da primeira instrução SQL e, em seguida, subtrai as que estão presentes na segunda instrução SQL de modo a obter o resultado final. Se a segunda instrução SQL incluir resultados que não estão presentes na primeira instrução SQL, esses resultados são ignorados.

A sintaxe será a seguinte:

```
[Instrução SQL 1]
MINUS
[Instrução SQL 2];
```

Vamos prosseguir com o mesmo exemplo:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Internet_Sales***

Txn_Date	Sales
07-Jan-1999	250
10-Jan-1999	535
11-Jan-1999	320
12-Jan-1999	750

e queremos descobrir todas as datas em que ocorreram vendas na loja, mas não através da Internet. Para tal, utilizamos a seguinte instrução SQL:

```
SELECT Txn_Date FROM Store_Information
MINUS
SELECT Txn_Date FROM Internet_Sales;
```

Resultado:

Txn_Date
05-Jan-1999
08-Jan-1999

'05-Jan-1999', '07-Jan-1999',e '08-Jan-1999' são valores diferentes obtidos através de **SELECT Txn_Date FROM Store_Information**. '07-Jan-1999' também é obtido através da segunda instrução SQL, **SELECT Txn_Date FROM Internet_Sales**, pelo que é excluído do conjunto de resultados finais.

Note que o comando **MINUS** apenas irá apresentar valores diferentes.

Algumas bases de dados podem utilizar **EXCEPT** em vez de **MINUS**. Consulte a documentação especifica sobre a sua base de dados para obter informações sobre uma utilização correta.

SQL >Linguagem SQL avançada >Subquery

É possível integrar uma instrução SQL noutra. Quando tal é efetuado nas instruções **WHERE** ou **HAVING**, possuem uma construção de consulta secundária.

A sintaxe será a seguinte:

```
SELECT "nome_coluna1"
FROM "nome_tabela1"
WHERE "nome_coluna2" [Comparison Operator]
(SELECT "nome_coluna3"
FROM "nome_tabela2"
WHERE "condição");
```

[Comparison Operator] pode ser um operador de igualdade como =, >, <, >=, <=. Também pode ser um operador de texto como "LIKE". A parte a **vermelho** é considerada como sendo uma "consulta interna", enquanto a parte **verde** é considerada como sendo a "consulta externa".

Vamos utilizar o mesmo exemplo conforme utilizado para ilustrar as uniões SQL:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

e pretendemos utilizar uma consulta secundária para descobrir as vendas de todas as lojas na região West. Para tal, utilizamos a seguinte instrução SQL:

```
SELECT SUM(Sales) FROM Store_Information
WHERE Store_Name IN
(SELECT Store_Name FROM Geography
WHERE Region_Name = 'West');
```

Resultado:

SUM(Sales)
2050

Neste exemplo, em vez de unir diretamente as duas tabelas e, em seguida, adicionar apenas o montante de vendas para as lojas na região West, primeiro utilizamos a consulta secundária para descobrir que lojas se encontram na região West e, em seguida, somamos o montante de vendas dessas lojas.

No exemplo acima, a consulta interna é executada em primeiro lugar e o resultado é então fornecido à consulta externa. Este tipo de consulta secundária denomina-se **consulta secundária simples**. Se a consulta interna estiver dependente da consulta externa, iremos obter uma **consulta secundária correlacionada**. Abaixo é apresentado um exemplo de uma **consulta secundária correlacionada**:

```
SELECT SUM(a1.Sales) FROM Store_Information a1
WHERE a1.Store_Name IN
(SELECT Store_Name FROM Geography a2
WHERE a2.Store_Name = a1.Store_Name);
```

Repare na cláusula **WHERE** na consulta interna, enquanto a condição envolve uma tabela da consulta externa.

SQL >**Linguagem SQL avançada** >**Exists**

Na secção anterior, utilizamos **IN** para ligar a consulta interna e a consulta externa numa instrução subsequente. **IN** não é a única forma de fazê-lo -- é possível utilizar muitos operadores como >, <, ou =. **EXISTS** é um operador especial que será abordado nesta secção.

EXISTS simplesmente testa se a consulta interna apresenta qualquer linha. Se apresentar, a consulta externa prossegue. Se não apresentar, a consulta externa não é executada e toda a instrução SQL não apresenta qualquer resultado.

A sintaxe de **EXISTS** é:

```
SELECT "nome_coluna1"
FROM "nome_tabela1"
WHERE EXISTS
(SELECT *
FROM "nome_tabela2"
WHERE "condição");
```

Note que em vez de *, pode selecionar uma ou mais colunas na consulta interna. O efeito será idêntico.

Vamos utilizar as mesmas tabelas como exemplo:

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabela ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

e emitimos a seguinte consulta SQL:

```
SELECT SUM(Sales) FROM Store_Information
WHERE EXISTS
(SELECT * FROM Geography
WHERE Region_Name = 'West');
```

Iremos obter o seguinte resultado:

SUM(Sales)

2750

No início, isto pode parecer confuso porque a consulta secundária inclui a condição [region_name = 'West'], embora a consulta tenha somado as lojas de todas as regiões. Após uma inspecção mais atenta, descobrimos que como a consulta secundária apresenta mais de 0 linhas, a condição **EXISTS** é verdadeira e a condição existente na consulta interna não influencia a forma como a consulta externa é executada.

SQL >Linguagem SQL avançada >Case

CASE é utilizado para fornecer o tipo de lógica "if-then-else" à linguagem SQL. A sua sintaxe é:

```
SELECT CASE ("nome_coluna")
  WHEN "condição1" THEN "resultado1"
  WHEN "condição2" THEN "resultado2"
  ...
  [ELSE "resultadoN"]
  END
FROM "nome_tabela";
```

"condição" pode ser um valor estático ou uma expressão. A cláusula **ELSE** é opcional.

No nosso exemplo, a Tabela ***Store_Information***,

Tabela ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999

San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

se quisermos multiplicar o montante de vendas de 'Los Angeles' por 2 e o montante de vendas de 'San Diego' por 1,5, escrevemos,

```
SELECT Store_Name, CASE Store_Name
  WHEN 'Los Angeles' THEN Sales * 2
  WHEN 'San Diego' THEN Sales * 1.5
  ELSE Sales
  END
'New Sales',
Txn_Date
FROM Store_Information;
```

'New Sales' é o nome dado à coluna com a instrução **CASE**.

Resultado:

Store_Name	New Sales	Txn_Date
Los Angeles	3000	05-Jan-1999
San Diego	375	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

SQL > Linguagem SQL avançada > NULL

Na linguagem SQL, **NULL** significa que os dados não existem. NULL não é igual a 0 ou uma cadeia vazia. Tanto 0 como uma cadeia vazia representam um valor, enquanto **NULL** não tem qualquer valor.

Todas as operações matemáticas efetuadas com **NULL** irão ter como resultado **NULL**. Por exemplo,

10 + NULL = NULL

As funções agregadas, tais como SUM, COUNT, AVG, MAX e MIN excluem os valores NULL. Não é provável que provoque quaisquer problemas com os comandos SUM, MAX e MIN. Contudo, pode provocar confusões com AVG e COUNT.

Observemos o seguinte exemplo:

Tabela ***Sales_Data***

Store_Name	Sales
Store A	300
Store B	200
Store C	100
Store D	NULL

Abaixo são apresentados os resultados de cada função agregada:

SUM (Sales) = 600

AVG (Sales) = 200

MAX (Sales) = 300

MIN (Sales) = 100

COUNT (Sales) = 3

Note que a função AVG conta apenas 3 linhas (a linha NULL é excluída), pelo que a média é 600 / 3 = 200 e não 600 / 4 = 150. A função COUNT também ignora a linha NULL, pelo que COUNT (Sales) = 3.

SQL > **Linguagem SQL avançada** > **Função ISNULL**

A função **ISNULL** encontra-se disponível nos sistemas SQL Server e MySQL. Contudo, as suas utilizações são diferentes:

SQL Server

No SQL Server, a função **ISNULL( )** é utilizada para substituir o valor NULL por outro valor.

Por exemplo, se tivermos a seguinte tabela,

Tabela ***Sales_Data***

Store_Name	Sales
Store A	300
Store B	NULL

A seguinte instrução SQL,

```
SELECT SUM ( ISNULL (Sales,100) ) FROM Sales_Data;
```

apresenta o valor 400. Isso deve-se ao fato de NULL ter sido substituído por 100 através da função ISNULL.

MySQL

No sistema MySQL, a função **ISNULL( )** é utilizada para testar se uma expressão é NULL. Se a expressão for NULL, esta função apresenta o valor 1. Caso contrário, esta função apresenta o valor 0.

Por exemplo,

ISNULL(3*3) apresenta o valor 0
ISNULL(3/0) apresenta o valor 1

SQL > **Linguagem SQL avançada** > **Função IFNULL**

A função IFNULL( ) encontra-se disponível no sistema MySQL, mas não nos sistemas SQL Server ou Oracle. Esta função requer dois argumentos. Se o primeiro argumento não for NULL, a função apresenta o primeiro argumento. Caso contrário, é apresentado o segundo argumento. Normalmente esta função é utilizada para substituir o valor NULL por outro valor. É semelhante à **função NVL**no sistema Oracle e à **função ISNULL** no SQL Server.

Por exemplo, se tivermos a seguinte tabela,

Tabela ***Sales_Data***

Store_Name	Sales
Store A	300
Store B	NULL

A seguinte instrução SQL,

```
SELECT SUM ( IFNULL (Sales,100)) FROM Sales_Data;
```

apresenta o valor 400. Isso deve-se ao fato de NULL ter sido substituído por 100 através da função **IFNULL**.

SQL > **Linguagem SQL avançada** > **Função NVL**

A função NVL( ) encontra-se disponível no sistema Oracle, mas não nos sistemas MySQL ou SQL Server. Esta função é utilizada para substituir o valor NULL por outro valor. É semelhante à **função IFNULL** no sistema MySQL e à **função ISNULL** no SQL Server.

Por exemplo, se tivermos a seguinte tabela,

Tabela ***Sales_Data***

Store_Name	Sales
Store A	300
Store B	NULL
Store C	150

A seguinte instrução SQL,

```
SELECT SUM ( NVL (Sales,100)) FROM Sales_Data;
```

apresenta 550. Isso deve-se ao fato de NULL ter sido substituído por 100 através da função **NVL**, pelo que a soma das 3 linhas é 300 + 100 + 150 = 550.

SQL > **Linguagem SQL avançada** > **Função Coalesce**

A função **COALESCE** na linguagem SQL apresenta a primeira expressão não-NULL entre os seus argumentos.

Passa-se o mesmo com a seguinte instrução **CASE**:

```
SELECT CASE ("nome_coluna")
  WHEN "expressão 1 is not NULL" THEN "expressão 1"
  WHEN "expressão 2 is not NULL" THEN "expressão 2"
  ...
  [ELSE "NULL"]
  END
FROM "nome_tabela";
```

Por exemplo, se tivermos a seguinte tabela,

Tabela ***Contact_Info***

Name	Business_Phone	Cell_Phone	Home_Phone
Jeff	531-2531	622-7813	565-9901
Laura	NULL	772-5588	312-4088
Peter	NULL	NULL	594-7477

e quisermos descobrir a melhor forma de contactar cada pessoa de acordo com as seguintes regras:

1. Se uma pessoa possuir um telefone da empresa, utilizar o número de telefone da empresa.

2. Se uma pessoa não possuir um telefone da empresa e possuir um telemóvel, utilizar o número do telemóvel.

3. Se uma pessoa não possuir um telefone da empresa, nem telemóvel, mas possuir telefone em casa, utilizar o número de telefone de casa.

É possível utilizar a função **COALESCE** para atingir o objetivo:

```
SELECT Name, COALESCE (Business_Phone, Cell_Phone, Home_Phone) Contact_Phone
FROM Contact_Info;
```

Resultado:

Name	Contact_Phone
Jeff	531-2531
Laura	772-5588
Peter	594-7477

SQL > **Linguagem SQL avançada** > **Função NULLIF**

A função **NULLIF** requer dois argumentos. Se os dois argumentos forem iguais, será obtido NULL. Caso contrário, é apresentado o primeiro argumento.

Passa-se o mesmo com a seguinte instrução **CASE**:

```
SELECT CASE ("nome_coluna")
  WHEN "expressão 1 = expressão 2 " THEN "NULL"
  [ELSE "expressão 1"]
  END
FROM "nome_tabela";
```

Por exemplo, suponhamos que temos uma tabela que regista as vendas actuais e o objectivo de vendas conforme abaixo:

Tabela ***Sales_Data***

Store_Name	Actual	Goal
Store A	50	50
Store B	40	50
Store C	25	30

Pretendemos mostrar NULL se as vendas actuais forem iguais ao objectivo de vendas e mostrar as vendas actuais se os dois valores forem diferentes. Para tal, utilizamos a seguinte instrução SQL:

```
SELECT Store_Name, NULLIF (Actual, Goal) FROM Sales_Data;
```

O resultado é:

Store_Name	NULLIF (Actual, Goal)
Store A	NULL
Store B	40
Store C	25