ORACLE

INTRODUÇÃO A SQL, PL/SQL E ADMINISTRAÇÃO DE BANCO DE DADOS COM ORACLE

Paulo Henrique Santos da Silva

2021

INTRODUÇÃO

Este artigo visa ser uma introdução ao aprendizado de SQL no banco de dados Oracle, PL/SQL e administração de bancos de dados Oracle. Aprendizado este, obtido nas aulas sobre o Oracle na Alura ministradas pelo instrutor Victorino Vila, e conteúdo da documentação Oracle e do site Oracle Tutorial. Ao final do artigo, nas referências bibliográficas, listarei alguns livros para se aprofundar no assunto, incluindo obras que abordam conceitos de banco de dados em geral, pois não tenho como objetivo apresentar neste artigo conceitos de normalização de banco de dados, álgebra relacional, DER, MER e afins.

O primeiro capítulo cobre o básico e essencial do SQL, como consulta de dados, ordenação e agrupamento de dados, filtros, *joins*, subconsultas, *views* entre outros comandos que fazem parte da DML (*Data Manipulation Language*). No segundo capítulo é abordado consultas mais inteligentes no Oracle com a PL/SQL, permeando assuntos como *procedures*, *functions*, *exceptions* e *packages*. E por fim, o terceiro capítulo é referente a administração de banco de dados Oracle, no qual exponho algumas atividades de um DBA Oracle, como criação e gerenciamento do banco, segurança e otimização do banco, análise do ambiente e otimização de consultas.

1 - SQL

A *Structured Query Language*, conhecida como SQL, surgiu na década de 1970 como uma linguagem projetada para gerenciar os dados estruturados e mantidos nos sistemas de administração de banco de dados (*Data Base Management System* - DBMS), tornando-se um padrão ISO em 1987.

Os comandos SQL são divididos em *Data Definition Language* (DDL), *Data Query Language* (DQL), *Data Manipulation Language* (DML) e *Data Control Language* (DCL). A maioria das operações que envolvem esses comandos SQL serão abordados no artigo. Iniciando pela DQL e DML, que permitem fazer a consulta e manipulação de dados nas tabelas, com os principais comandos sendo o SELECT, INSERT, UPDATE, DELETE e MERGE.

1.1 Consultando dados

Nesta seção será abordado o comando SELECT para a consulta de dados na base de dados Oracle. O comando SELECT, sendo uma especificação do SQL, tende ter a mesma sintaxe para todos os bancos de dados relacionais, com pequenas variações em cada implementação, como a estudada neste artigo, sendo voltada para a base de dados Oracle.

Para construir a consulta com o comando SELECT sempre pergunte a si mesmo “O que eu quero buscar?” e “De onde eu quero buscar?”. Com isso em mente, podemos começar a definir a sintaxe básica do comando, que é SELECT o que? FROM de onde? E sendo mais específico temos a seguinte sintaxe:

```
SELECT coluna_1, coluna_2... FROM nome_da_tabela;
```

Digamos que nossa base de dados possua uma tabela “Funcionários”, e desta tabela queremos saber o nome, sobrenome e data de nascimento de todos os funcionários. Para isso, podemos definir o seguinte comando com SELECT:

```
SELECT nome, sobrenome, dta_nascimento FROM funcionarios;
```

Também é possível retornar todos os dados da tabela (não recomendável) usando o operador "*" da seguinte forma:

```
SELECT * FROM funcionarios;
```

A consulta (com o primeiro exemplo) retornará o seguinte conjunto de dados:

NOME	SOBRENOME	DTA_NASCIMENTO
Letícia	Matos Cardoso	16/12/1993
Ana	Laura Lemes	28/08/1998
Alexandre	de Moraes Vasconcelos	05/02/1992
...	...	...

Podemos fazer a pergunta de "Como?" esses dados serão retornados ao usar outros comandos SQL, mas por enquanto observe que foi definido o que queríamos "nome, sobrenome e data de nascimento" e de onde queríamos "tabela Funcionários".

As consultas com o comando SELECT podem ser mais complexas, sendo apresentado nesses exemplos a sua forma básica. Em seguida veremos outras cláusulas que podem enriquecer o comando SELECT e tornar as consultas mais sofisticadas e elegantes.

1.2 Consultando, filtrando e agrupando dados

Agora veremos como aplicar condicionais, filtrar e agrupar os dados das consultas usando as cláusulas `WHERE`, `DISTINCT`, `AND`, `OR`, `IN`, `LIKE`, `BETWEEN`, `FETCH`, `IS NULL`, `ORDER BY` `GROUP BY` e `HAVING`.

Quando fazemos a consulta dos dados com o comando SELECT, é retornado um conjunto de linhas para as colunas indicadas. Com o WHERE podemos especificar condições para os dados que devem ser retornados da consulta, ou seja, podemos dizer "Como?" esse conjunto de dados deve ser retornado.

Então comecemos a fazer novamente as perguntas, "O que deve ser retornado?", "De onde?" e agora "Como devem ser retornados?". Aplicando essas perguntas a sintaxe, teremos algo como SELECT o que? FROM de onde? WHERE como? E sendo mais específico:

```
SELECT coluna_1, coluna_2
```

```
FROM nome_da_tabela
WHERE condicao;
```

Nos exemplos anteriores usamos a tabela de funcionários, mas agora queremos consultar os dados de alguns produtos. Primeiro vamos definir “o que?” e “de onde?” vamos consultar, depois dizemos “como?” será feita a consulta, filtrando as linhas que serão retornadas. Queremos saber o nome, o preço e a quantidade (o que?) de produtos (de onde?) que estão com menos de 300 unidades em estoque (como?). Agora podemos construir a consulta da seguinte maneira:

```
SELECT nome, preco, quantidade
FROM produtos
WHERE quantidade < 300;
```

Essa consulta retorna todos os produtos que tenham menos de 300 unidades disponíveis em estoque, como exibido na Tabela 2:

NOME	PRECO	QUANTIDADE
Resma de Papel A4	14.99	128
Pincel Redondo	9.70	287
Giz de Cera	25.50	96
...	...	...

Observe que após o WHERE foi usado um dos operadores de desigualdade (!=, < >) junto a coluna de “quantidade” para definir o filtro da consulta, indicando que deveria ser retornado apenas os produtos cuja quantidade é menor < que 300. Existem vários outros operadores, como os de comparação que podem compor a consulta, dizendo como as linhas devem ser retornadas. Na Tabela 3 é exibido os operadores disponíveis para o banco Oracle.

OPERADOR	DESCRIÇÃO
=	Igualdade
!=, < >	Desigualdade
>	Maior que
<	Menor que
>=	Maior ou igual

<=	Menor ou igual
IN	Igual a qualquer valor da lista de valores especificados
ANY/SOME/ALL	Compara com o valor de uma lista de valores ou de uma subconsulta
NOT IN	Que não seja igual a qualquer valor da lista de valores especificados

O NOT pode ser usado com outros operadores como negação, entretanto veremos os demais operadores depois de entendermos como funciona cada uma das cláusulas.

O WHERE pode ser usado em conjunto com outras cláusulas, como o AND e OR que são operadores lógicos, sendo conhecidos em algumas linguagens de programação como && (AND) e | | (OR).

Apesar de ser um exemplo bem simples, veremos como usar o AND na construção da *query*, para isso digamos que alguém pediu para conferir os produtos com estoque menor que 300 unidades e custando menos de R$15,00. Uma das maneiras de se fazer essa consulta é usando o WHERE com o AND:

```
SELECT nome, preco, quantidade
FROM produtos
WHERE quantidade < 300 AND preco < 15.00;
```

Nessa consulta primeiro é verificado todos os produtos que tenham a quantidade em estoque menor que 300 unidades (como visto anteriormente), depois é selecionado aqueles que tenham o seu preço menor que R$15,00 validando a condição imposta e retornando o seguinte resultado:

NOME	PRECO	QUANTIDADE
Resma de Papel A4	14.99	128
Pincel Redondo	9.70	287
...	...	...

Para o OR muda que não é necessário ambas as condições serem verdadeiras, apenas uma já valida a consulta. Então se nessa mesma consulta o AND for trocado pelo OR, “todos” os dados avaliados, podem ser retornados. E o retorno será de todos os produtos que tenham menos de 300 unidades (mesmo que custe mais que R$15,00) em estoque ou o preço seja menor que o especificado (mesmo que tenha mais de 300 unidades em estoque), pois como já

referido ao menos uma das condições deve ser verdadeira. E essa substituição resultaria no seguinte retorno:

NOME	PRECO	QUANTIDADE
Resma de Papel A4	14.99	128
Pincel Redondo	9.70	287
Giz de Cera	25.50	96
Lápis N2	2.75	421
Massa de Modelar Atóxica	18.90	152
...	...	...

A partir do OR podemos fazer uma associação com o IN em sua sintaxe, que diz basicamente o seguinte na consulta: retorne todas as linhas que tenham algum desses valores. E como isso se relaciona com o OR? Quando usamos o OR dizemos na consulta: eu quero que retorne isso ou aquilo, pode ser qualquer um do que estou especificando. Consegue notar a semelhança? A principal diferença é vista no IN, que aceita uma "lista" de valores. A sintaxe é a seguinte:

```
SELECT coluna_1, coluna_2...
FROM nome_da_tabela
WHERE coluna_1 IN ('valor1', 'valor2', 'valor3'...);
```

Agora foi informado o nome de alguns produtos e nos pediram para verificar o preço de cada um. Podemos fazer isso usando o IN, com a seguinte consulta:

```
SELECT nome, preco
FROM produtos
WHERE nome IN ('Giz de Cera', 'Mapa Mundi para colorir',
'Cola');
```

Sendo o retorno da consulta o seguinte resultado:

NOME	PRECO
Giz de Cera	25.50
Mapa-Múndi para colorir	56.80

Cola	3.70

E se quiséssemos fazer consulta usando, por exemplo, uma palavra que está na descrição de um produto ou as iniciais do sobrenome de um cliente como filtro? Para isso usamos o LIKE.

O LIKE permite fazer uma consulta com base em um padrão de caracteres como filtro. Pode ser as letras no início ou final de uma palavra e até mesmo a palavra inteira em um texto. Para filtrar palavras que coincidam com alguns caracteres que estejam no fim é necessário usar o operador "%" seguido pelos caracteres de filtro, o contrário pode ser feito para consultar palavras que comecem com os caracteres de filtro, ou seja, primeiro informe os caracteres e em seguida use o operador de percentagem.

Então, para consultar pelo início, use o operador no final, e para consultar pelo final use o operador no início. E para fazer uma consulta *insensitive case* (que ignore se está em maiúsculo ou minúsculo) pode usar as funções LOWER([nome_da_coluna]) e UPPER([nome_da_coluna]).

A seguir é apresentado a sintaxe de uma consulta com o LIKE.

```
SELECT coluna_1, coluna_2...
FROM nome_da_tabela
WHERE UPPER(nome_da_coluna) LIKE '%CARACTERES%';
```

Agora queremos saber o nome dos clientes que tem 'Silva' como sobrenome. Nessa consulta podemos fazer da seguinte maneira:

```
SELECT nome FROM clientes WHERE UPPER(nome) LIKE '%SILVA%;
```

E como resultado da consulta é retornado os seguintes clientes:

NOME
Abel Silva Castro
Valdeci da Silva

Observe que na construção dessa *query* foi usado a função UPPER() e em seguida o sobrenome também foi colocado em upper case, isso é necessário pois como citado anteriormente essa função coloca todos os caracteres da coluna passada como parâmetro em upper case, logo a "*string*" a ser buscada também deve estar em upper case para que possa ser possível realizar a comparação, e o mesmo ocorre ao usar a função LOWER().

No LIKE foi usado % no início e no fim da palavra, isso basicamente diz: "Busque todos os nomes que tenham 'Silva', não importa se é no começo, meio ou fim". Isso não acontece se usar o símbolo de percentagem no início ou fim. Caso use no início, dirá na sua *query*: "Busque apenas os nomes que tenham 'Silva' no final".

```
SELECT nome FROM clientes WHERE LOWER(nome) LIKE '%silva';
```

NOME
Valdeci da Silva

Ao trocar a posição do operador "%", movendo-o para o final, estamos dizendo que o retorno da consulta deve ser o nome das pessoas que começa com 'Silva', mas como é um sobrenome, neste caso, não irá retornar nenhuma linha.

E se ao fazer uma consulta você percebe que algumas linhas estão duplicadas, aparecendo por exemplo o mesmo nome várias vezes. Para resolver esse tipo de problema, podemos usar o DISTINCT, como o nome diz faz a distinção. Se na coluna xyz aparece o mesmo valor várias vezes, usando o DISTINCT será retornado apenas uma ocorrência desse mesmo valor.

Primeiro segue a sintaxe do DISTINCT e logo em seguida um exemplo simples que mostrar como ele é muito útil nas consultas.

```
SELECT DISTINCT coluna_1... FROM nome_da_tabela;
```

Agora digamos que na tabela clientes também e armazenado os dados sobre a sua localidade, como endereço, cidade, estado, bairro etc. E alguém pediu para descobrir em quantos bairros da cidade de São Paulo estão os clientes da empresa. Primeiro vamos fazer

uma consulta sem usar o DISTINCT (e o COUNT) para ver como é o retorno, e em seguida usando o DISTINCT.

SELECT bairro FROM clientes WHERE cidade = ‘São Paulo’;

Essa query nos retorna o seguinte conjunto de dados.

BAIRRO
Jardins
Lapa
Santo Amaro
Bras
Jardins
Jardin

Observe que o bairro ‘Jardins’ aparece 3 vezes, isso significa que a empresa possui 3 clientes que moram nesse bairro. Entretanto não é isso que nos foi pedido, pois se contarmos todos esses bairros, incluindo os repetidos, entregaremos uma resposta equivocada em que incluímos o mesmo bairro 3 vezes na contagem. Para resolver isso pode ser usado o DISTINCT.

SELECT DISTINCT bairro FROM tabela_de_clientes WHERE cidade = ‘São Paulo’;

E agora é retornado os bairros, mas sem repetição.

BAIRRO
Jardins
Lapa
Santo Amaro
Bras

Então os clientes da empresa, na cidade de São Paulo, estão distribuídos em 4 bairros diferentes.

Até o momento todas as consultas retornadas estão “desordenadas”, aparecendo conforme a ordem de inserção (e operações) no banco de dados. Podemos ordenar as

consultas para aparecerem em ordem crescente (A-Z ou 1-10...) ou decrescente (Z-A ou 10-1). Para isso usamos o ORDER BY [ASC | DESC] e especificamos por qual coluna queremos ordenar.

```
SELECT coluna_1, coluna_2...
FROM nome_da_tabela
ORDER BY coluna_1 [ASC|DESC];
```

Como exemplo de consulta aos dados podemos usar a tabela de clientes, ordenando o resultado da consulta pelo nome em ordem crescente.

SELECT cpf, nome FROM clientes ORDER BY nome ASC;

Nessa *query* dizemos: 'Por obséquio, poderias buscar todos os dados de cpf e nome dos clientes, retornando-os em ordem crescente?'. E como resultado temos:

CPF	NOME
50534475787	Abel Silva
87196557702	Carlos Eduardo
26005867093	Cesar Teixeira
92837607944	Edson Meilelles
49247271856	Eduardo Jorge
...	...

Então podemos ordenar o resultado da consulta usando ORDER BY e se quiséssemos agrupar os dados? Para isso existe o GROUP BY! O GROUP BY, basicamente, agrupa os dados quantitativos em relação aos dados qualitativos. É muito comum usar o GROUP BY com funções de agregação, como COUNT(), SUM(), AVG(), MAX(), MIN().

```
SELECT coluna_1, coluna_2
FROM nome_da_tabela
GROUP BY colunaX;
```

E pode ser usado com outras clausulas, como WHERE ou ORDER BY, mas lembrando que o GROUP BY deve aparecer antes do ORDER BY.

```
SELECT coluna_1, coluna_2
FROM nome_da_tabela
WHERE condicao
GROUP BY colunaX
ORDER BY colunaXY;
```

O COUNT() permite contar o número de linhas retornados de uma determinada coluna para aquela consulta, ou seja, é uma função que faz a contagem de itens pertencentes a um grupo.

Podemos usar como exemplo a consulta realizada anteriormente para saber em quantos bairros os clientes da empresa estavam presentes. Nessa consulta fizemos a contagem observando o retorno, mas e se fosse muitas linhas retornadas? Pois bem, agora vamos usar o COUNT() na mesma consulta para realizar essa tarefa.

```
SELECT COUNT(DISTINCT bairro) AS Bairros
FROM clientes
WHERE cidade = 'São Paulo';
```

Observe que nessa *query* usamos o 'AS', isso serve para apelidar as colunas, digamos que por algum motivo uma coluna foi nomeada como 'gato_cachorro', mas na consulta para facilitar a leitura você queira chamá-la de 'pets', então use o 'AS' na frente do nome da coluna para apelidar 'gato_cachorro' como 'pets'. Esse tipo de alteração não é refletido na estrutura presente no banco de dados, servindo apenas como um apelido no momento de identificar o nome da coluna na consulta.

BAIRROS
4

Em relação ao retorno da consulta, note que não foi feita a contagem (de forma distinta) do número de bairros e retornado esse valor, sendo o oposto da consulta anterior, e muito mais eficiente nos casos de grandes consultas.

Agora vamos fazer uma consulta semelhante, mas dessa vez queremos saber o número de bairros por cidade, ou seja, vamos agrupar a contagem de bairro por cidade.

```
SELECT cidade, COUNT(DISTINCT bairro) AS ‘Bairros’
FROM clientes
GROUP BY cidade;
```

E como resultado temos o seguinte retorno da consulta:

CIDADE	BAIRROS
São Paulo	4
Rio de Janeiro	7
...	...

Agora imaginemos o seguinte cenário: cada cliente possui um limite de crédito que pode ser usado para compras na empresa e alguém nos pediu para descobrir qual estado possui o maior limite de crédito total.

Para entregar o que foi pedido (desprezando a otimização, assunto que será abordado nos próximos capítulos) podemos usar a função SUM() para somar o limite de credito de todos os clientes de cada estado e então agrupar essa soma por estado, finalizando com a ordenação por ordem decrescente do limite de crédito.

```
SELECT estado, SUM(limite_credito) AS Credito
FROM clientes
GROUP BY estado
ORDER BY Credito DESC;
```

A partir dessa consulta descobriremos qual estado possui o maior limite de crédito, como solicitado.

ESTADO	CREDITO
RJ	995000
SP	810000
...	...

Como podemos observar o estado do Rio de Janeiro tem um total de R$995.000,00 de crédito total.

E se desse estado quisermos saber qual é a média de crédito, o cliente que possui maior limite e o cliente com menor limite? Para isso podemos usar as funções AVG(), MAX() e MIN().

A função AVG() encontra a média (média aritmética simples) de um conjunto de valores.

```
SELECT AVG(coluna_1), coluna_2
FROM nome_da_tabela
GROUP BY colunaX
ORDER BY colunaY;
```

Aplicando essa sintaxe a nossa *query* podemos descobrir o valor médio de crédito por estado.

```
SELECT estado, AVG(limite_credito) AS Credito
FROM clientes
GROUP BY estado
ORDER BY Credito DESC;
```

Como resultado da consulta obtemos a média para cada estado.

ESTADO	CREDITO
RJ	110555,55
SP	135000
...	...

Determinaremos agora qual cliente possui maior e menor limite de credito do estado de São Paulo, usando a função MAX() para encontrar quem possui o maior limite e a função MIN() para encontrar quem possui o menor.

```
SELECT coluna_1, MAX(coluna_2)
FROM nome_da_tabela
GROUP BY colunaX
```

```
ORDER BY colunaY;
```

Na construção da *query* para encontrar o maior limite fica da seguinte maneira.

```
SELECT nome, MAX(limite_credito) AS Credito
FROM clientes
WHERE estado = 'SP'
GROUP BY nome
ORDER BY Credito DESC;
```

Desconsiderando o uso de *subqueries* para filtrar a consulta, obtemos como resultado o limite de crédito dos clientes em ordem decrescente, sendo possível observar qual possui maior limite.

NOME	CREDITO
Carlos Eduardo	200000
Erica Carvalho	170000
...	...

Para encontrar o menor valor poderíamos apenas inverter a ordenação, mas vamos ver como a sintaxe se aplica a função MIN() para descobrir o menor valor.

```
SELECT coluna_1, MAX(coluna_2)
FROM nome_da_tabela
GROUP BY colunaX
ORDER BY colunaY;
```

Aplicando a sintaxe na construção da *query*.

```
SELECT nome, MIN(limite_credito) AS Credito
FROM clientes
WHERE estado = 'SP'
GROUP BY nome
```

```
ORDER BY Credito ASC;
```

Agora podemos descobrir qual cliente possui o menor limite de crédito do estado de São Paulo.

NOME	CREDITO
Carlos Eduardo	200000
Erica Carvalho	170000
...	...

Para finalizar com o GROUP BY, vamos entender como funciona o HAVING, que é o filtro dos grupos. Em essência, o HAVING filtra um grupo de linhas, e por isso geralmente é usado com o GROUP BY. O seu uso é semelhante ao do WHERE, mas antes você deve ter um conjunto de dados ao qual vai aplicar uma condição.

```
SELECT coluna_1, coluna_2...
FROM nome_da_tabela
GROUP BY colunaX
HAVING condicao;
```

Dessa vez precisamos saber quais foram os clientes que fizeram mais de 100 compras no ano de 2016. Para isso podemos consultar a tabela de notas fiscais, contar o número de compras feitas, determinar a data das compras, agrupar por CPF e filtrar pela quantidade requisitada.

```
SELECT cpf, COUNT(*) AS Compras
FROM notas_fiscais
WHERE TO_CHAR(data_venda, 'YYYY') = '2016'
GROUP BY cpf
HAVING COUNT(*) > 100;
```

Na construção dessa *query* usamos uma função que faz a conversão de datas (como um toString, parse etc das linguagens de programação) para o tipo string (VARCHAR2 no

Oracle), o TO_CHAR() para que então possamos comparar, sem dificuldades, com a data que quisermos. E assim temos todos os clientes que fizerem mais de 100 compras em 2016.

CPF	Compras
3623344710	162
492472718	123
...	...

Com essa última clausula, criamos uma base para a próxima seção que decorrera sobre junção entre tabelas e *subqueries*.

1.3 Consultas com junção entre tabelas, *subqueries* e *views*

Nesta seção vamos começar a anteder conceitos básicos de junções entre tabelas e *subqueries* avançando no decorrer dos próximos capítulos sobre a eficiência em consultas com esse tipo *query*.

No banco de dados existe diferentes tipos de junção entre tabelas, conhecidos como `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN` e `FULL JOIN`, que usam de conceitos da álgebra relacional.

Começaremos pelo INNER JOIN, observe a Figura 1.

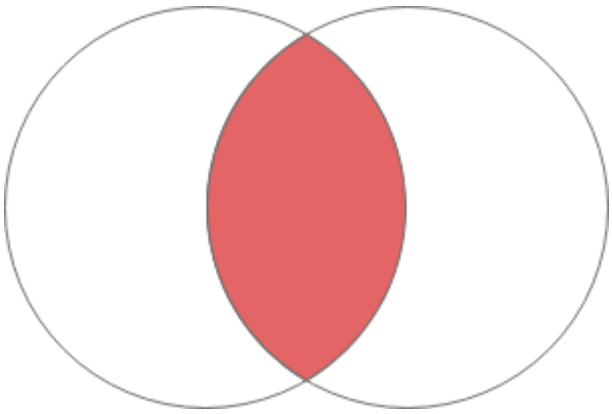

Figura 1 - INNER JOIN

A Figura 1 é uma representação da intersecção de dois conjuntos em que a intersecção denota os elementos que pertencem, simultaneamente, a ambos os conjuntos. Em

outras palavras, os elementos do conjunto a esquerda pertencem aos elementos do conjunto a direita e vice-versa.

Na junção entre tabelas, está é a representação do INNER JOIN, com ele consultaremos os dados da tabela B que fazem referência a tabela A, sendo muito importante ter o conhecimento de modelagem de dados, pois para realizar a junção das tabelas, usaremos conceitos de chave primaria e chave estrangeira.

Quando fazemos o INNER JOIN ele vai buscar somente pela chave estrangeira dos dados de uma tabela que fazem referência aos dados de outra tabela. A sintaxe do INNER JOIN é representada da seguinte maneira:

```
SELECT coluna_1, coluna_2...
FROM nome_da_tabela_A
INNER JOIN nome_da_tabela_B
ON nome_da_tabela_A.pk = nome_da_tabela_B.fk;
```

Em que pk (*primary key*) representa a chave primaria da tabela A e fk (*foreign key*) representa a chave estrangeira na tabela B que faz referência à tabela A.

Na última consulta, usando o HAVING, buscamos pelos clientes que compraram mais de 100 produtos em 2016, e como retorno obtivemos o CPF dos clientes e a quantidade que compraram nessa data, mas não vimos os nomes desses clientes e nem sabemos de onde eles são, pois esses dados pertencem a tabela de clientes e não de notas ficais.

Entretanto, observe que na tabela de notas fiscais o CPF é a chave estrangeira que faz referência a tabela de clientes. Sabendo disso, podemos fazer uma junção entre essas duas tabelas e obter mais dados a respeito do cliente, como a sua data de nascimento o estado ao qual pertence.

```
SELECT COUNT(*), nf.cpf, c.dta_nascimento, c.estado
FROM notas_fiscais nf
INNER JOIN clientes c
ON nf.cpf = c.cpf
WHERE TO_CHAR(data_venda, 'YYYY')  = '2016'
GROUP BY nf.cpf, c.dta_nascimento, c.estado
```

```
HAVING COUNT(*) > 100;
```

Com a junção dos dados de ambas as tabelas podemos saber de onde nossos clientes são e quando nasceram, talvez essa seja uma informação relevante para o departamento de marketing.

COMPRAS	CPF	DTA_NASCIMENTO	ESTADO
162	3623344710	13/01/95	RJ
123	492472718	19/07/94	RJ
...	...	...	...

Na construção de *queries* usando o JOIN é comum apelidar as tabelas, como no exemplo acima em que a tabela de notas fiscais foi apelidada como “nf” e a tabela de clientes como “c”. Dessa forma a consulta fica mais legível, apenas não exagere em resumir o nome das tabelas a um apelido que não faça sentido, seja coerente.

Outra maneira de fazer a junção entre as tabelas, é consultando pelos dados que se relacionam, mas pertencem a tabela A ou tabela B. Nesse caso estamos falando do RIGHT JOIN e do LEFT JOIN, que além dos dados da intersecção, buscam os dados da tabela ao lado direito da consulta ou os dados da tabela ao lado esquerdo da consulta.

Para ilustrar melhor, observe a Figura 2, que representa o RIGHT JOIN.

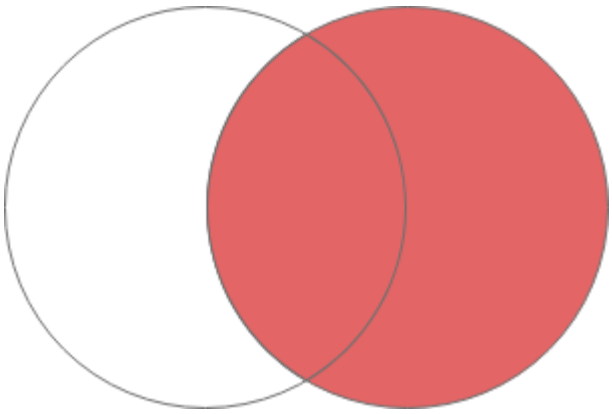

Figura 2 - RIGHT JOIN

Quando usamos o INNER JOIN é retornado para a consulta somente os dados que se relacionam, caso alguns dados da tabela A não tenha relação com a tabela B essa linha da consulta não será retornada. No entanto, gostaríamos de ver além desses dados, incluindo os

dados de uma tabela ou outra. O RIGT JOIN nos permite fazer isso, consultar os dados que pertence somente a ambos os conjuntos mais os dados da tabela a direta.

```
SELECT coluna_1, coluna_2...
FROM nome_da_tabela_esquerda
RIGHT JOIN nome_da_tabela_direita
ON nome_da_tabela_esquerda.pk = nome_da_tabela_direita.fk
```

Sabemos que muitas pessoas faz o seu cadastro nos sites de compras, mas não realizam nenhuma compra. Então foi pedido para identificarmos na nossa base de dados quem são esses clientes, que tem cadastro, mas nunca compraram.

Podemos pensar da seguinte maneira, a pessoa vai estar na tabela cliente, pois ela fez o cadastro, mas se ela não fez nenhuma compra, então não vai estar na tabela de notas fiscais. Sabendo disso, podemos fazer um RIGHT (ou LEFT) JOIN entre essas tabelas, e será retornado todos os dados de uma ou outra, mas ficará faltando a referência (NULL) dos que não fizeram compras.

```
SELECT DISTINCT c.cpf AS Cpf_Cliente, c.nome AS Cliente,
nf.cpf AS Cpf_Nota
FROM clientes c
RIGHT JOIN notas_fiscais nf
ON c.cpf = nf.cpf;
```

Nessa consulta vamos trazer os dados que tem relação entre si mais os dados de notas fiscais (RIGHT JOIN - ao lado direito), e observaremos a ausência de uma referência em notas fiscais.

CPF_CLIENTE	CLIENTE	CPF_NOTA
14711567107	Erica Carvalho	14711567107
55762287582	Petra Oliveira	55762287582
19290992743	Fernando Cavalcante	null
...	...	...

O “null” indica que este cliente está cadastrado em nossa base de dados, mas não fez nenhuma compra e, como sabemos o que esse null indica, poderíamos ainda filtrar a consulta apenas pelos que apresentam null na tabela de notas fiscais.

E muito semelhante ao RIGHT JOIN, diferindo apenas em qual lado vai buscar os dados, podemos representar o LEFT JOIN como no diagrama da Figura 3.

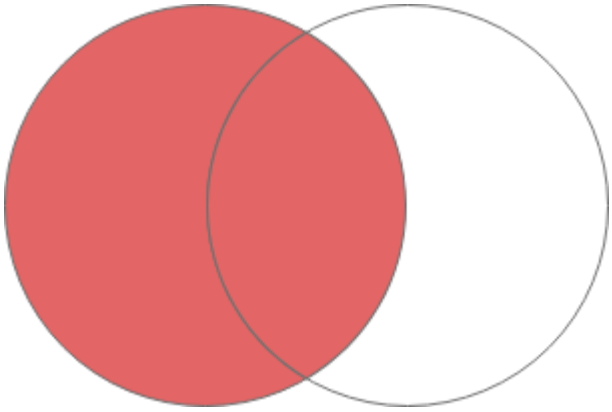

Figura 3 - LEFT JOIN

O LEFT JOIN possui a seguinte sintaxe:

```
SELECT coluna_1, coluna_2...
FROM nome_da_tabela_esquerda
LEFT JOIN nome_da_tabela_direita
ON nome_da_tabela_esquerda.pk = nome_da_tabela_direita.fk
```

Podemos fazer a mesma consulta, mas invertendo apenas a ordem de obtenção dos dados usando o LEFT JOIN.

```
SELECT DISTINCT c.cpf AS Cpf_Cliente, c.nome AS Cliente,
nf.cpf AS Cpf_Nota
FROM notas_fiscais nf
LEFT JOIN clientes c
ON c.cpf = nf.cpf
WHERE nf.cpf = NULL;
```

Essa *query* com o LEFT JOIN busca os dados das notas fiscais, que dessa vez é a tabela da esquerda, após o FROM, e filtra para trazer apenas os clientes que não fizerem compras.

CPF_CLIENTE	CLIENTE	CPF_NOTA
19290992743	Fernando Cavalcante	null
...	...	...

Podemos consultar os dados na tabela a esquerda com o LEFT JOIN, a direita com o RIGHT JOIN ou apenas os que tem relação entre si com o INNER JOIN. Para trazer os dados de ambas as tabelas, podemos usar o FULL JOIN, como apresentado na Figura 4.

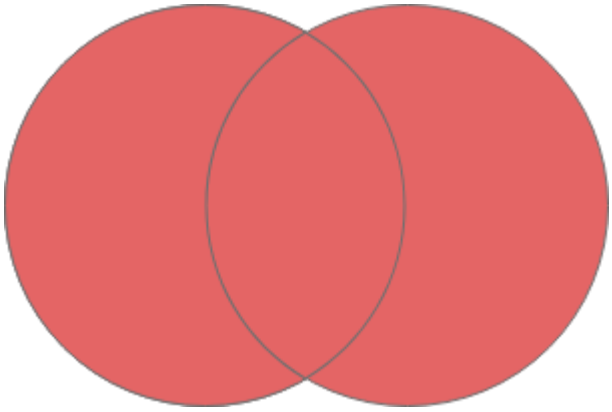

Figura 4 - FULL OUTER JOIN

O FULL JOIN retornará todos os dados que um INNER JOIN, RIGHT JOIN e LEFT JOIN, combinando-os na mesma clausula, ou seja, busca todos os dados que se relacionam e ainda todos os outros dados que não se relacionam. Os dados que não se relacionam são exibidos como null no retorno da consulta.

Uma outra maneira de juntar as tabelas é usando o UNION e UNION ALL. Assim como o INNER JOIN o UNION fará a relação entre as tabelas, retornado os dados que estão relacionados. Entretanto, é importante observar que as colunas devem ser do mesmo tipo de dado e o número de coluna no primeiro SELECT deve ser o mesmo para o segundo SELECT.

```
SELECT coluna_1, coluna_2
FROM nome_da_tabela_1
UNION
```

```
SELECT coluna_1, coluna_2
FROM nome_da_tabela_2;
```

Nesse banco de dados, além dos clientes, também é cadastrados os vendedores. Usando o UNION podemos saber se existe clientes e vendedores que moram no mesmo bairro.

```
SELECT bairro, CLIENTE AS 'TIPO'
FROM clientes
UNION
SLECT bairro, VENDEDOR AS 'TIPO'
FROM vendedores;
```

Com essa *query* é retornado o seguinte resultado:

BAIRRO	TIPO
Copacabana	VENDEDOR
Jardins	VENDEDOR
Jardins	CLIENTE
...	...

Observe que tivemos duas ocorrências no mesmo bairro, mostrando cliente e vendedor.

A diferença entre o UNION e UNION ALL está no DISTINCT implícito do UNION que permite fazer a filtragem de linhas repetidas, diferindo do UNION ALL que retorna exatamente todas as linhas.

Uma outra maneira de consultar e filtrar os dados é usando *subqueries*. Basicamente é uma *query* "dentro" de outra *query*, sendo a consulta interna usada para retornar um conjunto de dados e a consulta externa usada para filtrar esse conjunto de dados.

Uma exemplificação simples da sintaxe de uma *subquery* com WHERE pode ser visto a seguir.

```
SELECT coluna_1, coluna_2
FROM nome_da_tabela_1
WHERE condicao =
(
     SELECT coluna_1, coluna_2
     FROM nome_da_tabela_2
     WHERE condicao
);
```

Anteriormente usamos a função MAX() para descobrir qual cliente possui o maior limite de crédito. Agora poderíamos usar essa função em conjunto com uma *subquery* para descobrir qual produto da empresa é o mais caro.

```
SELECT código, nome, preco
FROM produtos
WHERE preco =
(
    SELECT MAX(preco)
    FROM produtos
);
```

Com essas *subquery* podemos obter o maior preço dos nossos produtos, então com a *query* mais externa conseguimos filtrar esse valor e obter o seu código e nome, agregando mais informação a nossa consulta e descobrindo qual produto é o mais caro.

CODIGO	NOME	PRECO
1022450	Parker Caneta-tinteiro, com acabamento cromado	1000,69

Uma linda caneta Parker no valor de R$1.000,69!

Obviamente consultas mais complexas podem ser feitas usando *subqueries*, entretanto é sempre usar esse recurso com cuidado, visto que é retornado um conjunto de dados que ainda será filtrado ou aplicado outra operações.

Algo ainda muito interessante e cômodo é as VIEWS. Uma VIEW é a construção de uma tabela virtual a partir de uma *query*. Digamos que precisamos construir uma query grande e complexa, mas não queremos ter que ficar fazendo isso a todo momento. Para solucionar esse problema, podemos fazer a query uma única vez e a partir dela construir uma VIEW e sempre que quisermos visualizar os dados dessa consulta ou fazer manipulações, podemos apenas chamar essa VIEW.

```
CREATE VIEW nome_view AS (
    SELECT coluna_1, coluna_2...
    FROM nome_da_tabela
);
```

Podemos usar como exemplo a consulta anterior, criada para exemplificar o uso de *subqueries*.

```
CREATE VIEW vw_maxpreco AS (
      SELECT código, nome, preco
      FROM produtos
      WHERE preco =
      (
         SELECT MAX(preco)
         FROM produtos
      )
);
```

Quando acessar a guia “Views” no SQLDeveloper, poderá ver a VIEW que acabou de criar, como exibido na Figura 5.

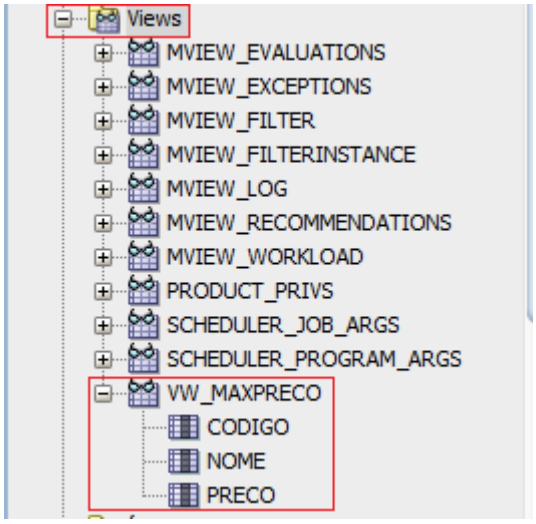

Figura 5 - VW_MAXPRECO

A partir dessa VIEW poderíamos fazer outras consultas, e descobrir por exemplo quais clientes adquiriram este produto e de onde eles são. E após utilizar a VIEW, caso seja necessário, podemos apagá-la usando um comando DROP.

```
DROP VIEW nome_da_view;
```

Na próxima seção será discutido algumas funções Oracle para manipulação dos dados, como funções matemáticas e de datas.

1.4 Funções matemáticas, manipulação de *strings*, datas e conversão de dados

Nesta seção abordaremos algumas funções Oracle para manipulação de dados, como funções matemáticas, funções para manipulação de *strings*, datas e conversão de dados. E levando em consideração as inúmeras funções presente nesse banco de dados, será abordado as mais usuais e básicas.

Funções para manipulação de *strings*

Função	Descrição
UPPER	Converte a *string* para letras maiúsculas
LOWER	Converte a *string* para letras minúsculas
INITCAP	Primeira letra de cada palavra fica maiúscula
CONCAT	Concatena duas *strings*

SUBSTR	Extrai parte da *string* a partir de uma posição

Duas dessas funções já foram usadas, a UPPER e a LOWER. A função UPPER converte todos os dados de uma coluna do tipo *string* (NVACHAR etc.) para *upper case*, deixando em maiúsculas. A função LOWER faz o inverso, pois converte todas as ocorrências para *lower case*, ficando em minúsculas. Como já foram usadas, então não serão exemplificadas.

A função INITCAP faz com que a primeira letra de cada palavra em um texto fique em maiúscula. A sua sintaxe é bem simples

```
SELECT INITCAP(coluna_1), coluna_2
FROM nome_da_tabela;
```

Podemos aplicar essa função na coluna de nomes da tabela de clientes, colocando cada palavra do nome com as iniciais em maiúscula, ficando mais legível e agradável para a visualização.

```
SELECT nome AS Antes, INITCAP(nome) AS Depois
FROM clientes;
```

Para compara, como um antes e depois, podemos selecionar a coluna “nome” com a função e sem a função, obtendo o seguinte resultado.

ANTES	DEPOIS
Erica carvalho	Erica Carvalho
fernando Cavalcante	Fernando Cavalcante
Cesar Teixeira	Cesar Teixeira
Marcos nogueira	Marcos Nogueira
eduardo Jorge	Eduardo Jorge
...	...

Outra função interessante para manipular *string* é o CONCAT, que pode ser substituído por sua “sintaxe mais curta” que são dois pipes (||). O CONCAT junta duas *strings* em uma única *string*.

```
SELECT coluna_1 || ‘ ’|| coluna_2
FROM nome_da_tabela;
```

Podemos concatenar as colunas que representam o endereço do cliente, formando uma única *string*.

```
SELECT nome, ‘Endereco: ’ || endereço || ‘ ’ || bairro ||
‘ ’ || cidade ||‘ ’|| estado AS Endereco
FROM clientes;
```

E com isso obtemos uma única coluna de endereço para o cliente.

NOME	ENDERECO
Paulo Cesar Mattos	Endereço: R. Srg. Edison de Oliveira Jardins Sao Paulo SP
Gabriel Araujo	Endereço: Av. Gen. Guedes da Fontoura Jardins Sao Paulo SP
Marcelo Mattos	Endereço: R. Moraes de Azevedo Cidade Nova Rio de Janeiro RJ
...	...

A função usada para dividir uma *string* em outras *strings* é a SUBSTR. Essa função aceita três argumentos como parâmetros, sendo o primeiro o nome da coluna, o valor que represente a partir de qual posição você deseja começar a dividir a *string* e o segundo parâmetro o valor de quantos caracteres devem ter a nova *string*.

```
SELECT SUBSTR(coluna_1, 3, 2)
FROM nome_da_tabela;
```

Usaremos como exemplo a coluna nome da tabela de clientes.

```
SELECT nome, SUBSTR(nome, 1, 7)
FROM clientes;
```

Nessa *query* pedimos para ser consultado os nomes da tabela de clientes e que os nomes fossem divididos a partir do primeiro caractere, pegando os 7 primeiros para a nova *string*.

NOME	SUBSTR(NOME, 1, 7)
Petra Oliveira	Petra O
Paulo Cesar Mattos	Paulo C
Gabriel Araujo	Gabriel
...	...

Funções para manipulação de datas.

Outras funções interessantes para se ter conhecimento, é sobre as que manipulam datas, sendo possível até mesmo realizar algumas operações aritméticas simples, como adição e subtração. No banco de dados Oracle as datas seguem o calendário juliano e são armazenadas como números.

Função	Descrição
SYSDATE	Retorna a data do servidor
ADD_MONTHS	Adiciona *n* meses a data
MONTHS_BETWEEN	Retorna o número de meses entre duas datas
NEXT_DAY	Retorna a data seguinte à especificada
LAST_DAY	Retorna o último dia do mês da data atual

Para sabermos a data atual pelo sistema, podemos usar a função SYSDATE.

```
SELECT SYSDATE AS DATA FROM DUAL;
```

A tabela DUAL é a representação de uma tabela nula no banco de dados Oracle, muito utilizada para fazer testes que não precisam retornar dados, como para o uso de algumas funções. Nesse caso usamos a função SYSDATE que retorna a data atual.

DATA
27/06/21

Com essa data podemos fazer algumas operações matemáticas, como a adição e subtração.

```
SELECT SYSDATE + 1 AS DATA FROM DUAL;
```

DATA
28/06/21

E subtraindo a data

```
SELECT SYSDATE - 1 AS DATA FROM DUAL;
```

DATA
26/06/21

É importante destacar que a data adicionada ou subtraída é feita em dias, então se adicionar 15, serão adicionados 15 dias a data. Se quisermos adicionar meses a data, podemos usar a função ADD_MONTHS.

```
SELECT ADD_MONTHS(SYSDATE, 2) AS DATA FROM DUAL;
```

Nessa *query* adicionamos 2 meses a data atual.

DATA
27/08/21

Agora queremos descobrir o número de meses entre duas datas usando a função MONTHS_BETWEEN.

```
SELECT MONSTHS_BETWEEN(SYSDATE, TO_DATE('2021-01-01',
'YYYY-MM-DD')) AS DATA FROM DUAL;
```

Recebemos como retorno dessa consulta, 7 meses.

DATA
7,838709677419354838709677419354838709 68

Para sabermos qual será a data de um próximo dia, podemos usar a função NEXT_DAY e especificar qual dia queremos saber a partir de uma data.

```
SELECT NEXT_DAY(SYSDATE, 'SEXTA') AS DATA FROM DUAL;
```

DATA
02/07/21

Sendo a data atual 26/08/21 um domingo, a próxima sexta-feira será no dia 02/07/2021. E para saber o último dia do mês, usamos a função LAST_DAY informando uma data.

```
SELECT LAST_DAY(SYSDATE) AS DATA FROM DUAL;
```

DATA
30/06/21

Funções matemáticas.

O banco de dados Oracle possui diversas funções para realizar operações matemáticas, mas devido a esse grande número, veremos apenas as mais comuns.

Função	Descrição
ROUND	Arredonda um valor
TRUNC	Remove o decimal e deixa apenas o inteiro
FLOOR	Sempre arredonda para baixo
POWER	Aplica a potenciação ao valor
SQRT	Aplica a radiciação ao valor
MOD	Retorna o resto da divisão
ABS	Retorna o valor absoluto

Comecemos pelo ROUND que é a função utilizada para arredondar um valor.

```
SELECT ROUND(2.7) FROM DUAL;
```

ROUND(2.7)
3

Em contrapartida o TRUNC é a função que remove a parte decimal e deixa apenas o número inteiro.

```
SELECT TRUNC(14.52) FROM DUAL;
```

TRUNC(14.52)
14

Enquanto o ROUND arredonda de acordo com a proximidade do valor, o FLOOR sempre arredonda para baixo.

```
SELECT FLOOR(9.9999) FROM DUAL;
```

FLOOR(9.9999)
9

Aplicara a potenciação em um valor é muito simples, podemos usar o POWER para isso, passando primeiro a base e depois o expoente.

```
SELECT POWER(5, 3) FROM DUAL;
```

POWER(5, 3)
125

A operação inversa da exponenciação é a radiciação, expressa em função como SQRT.

```
SELECT SQRT(16) FROM DUAL;
```

SQRT(16)
4

O resto de uma divisão pode ser encontrado usando a função MOD.

```
SELECT MOD(16, 2) FROM DUAL;
```

MOD(16)
0

E se quisermos saber o módulo de um valor, isto é, o seu valor absoluto, sem sinais. Usemos a função ABS para resolver este problema.

```
SELECT ABS(-78) FROM DUAL;
```

ABS(-78)
78

Funções para conversão de dados.

Para finalizar esse tópico, veremos algumas funções para conversão de dados.

Função	Descrição
TO_DATE	Converte uma *string* para o tipo *date*
TO_CHAR	Converte um tipo *date* para o tipo *string*
TO_NUMBER	Converte um valor em *string* para numérico
NVL	Converte valores nulos para outro valor

A conversão de uma data no formato literal para o formato do tipo *date* pode ser feito usando a função TO_DATE que requer a data que deseja converter e o formato dessa data (máscara).

```
SELECT TO_DATE('25/12/2021', 'DD/MM/YYYY') AS DATA FROM
DUAL
```

Essa query retorna a data do Natal no formato que conhecemos no Brasil, começado pelo dia, depois o mês e então o ano.

DATA
25/12/21

Podemos passar essa data para o formato norte americano ao especificar a máscara.

```
SELECT TO_DATE('25/12/2021', 'MM/DD/YYYY') AS DATA FROM
DUAL;
```

DATA
12/25/21

E para fazer o inverso, converter do tipo *date* para o tipo *string* podemos usar a função TO_CHART, que já foi exibida em alguns exemplos anteriores. Devido a isso, veremos o TO_NUMBER, uma função que transforma um tipo literal para um tipo numérico.

```
SELECT TO_NUMBER('10') / 2 AS NUMERO FROM DUAL;
```

NUMERO
5

1.5 Introdução aos comandos de *Data Manipulation Language* e *Data Definition Language*

Nesta seção será abordado os conceitos básicos para a definição da estrutura de armazenamento dos dados no banco e a manipulação desses dados, ressaltando novamente que aqui será descrito o básico, pois com esses comandos você pode construir e destruir um banco de dados.

Os comandos especificados na *Data Manipulation Language* (DML) são utilizados para a manipulação de dados nas tabelas, sendo esses comandos o INSERT, SELECT, UPDATE e DELETE.

Para a definição e manipulação da estrutura de armazenamento dos dados é utilizado os comandos da *Data Definition Language* (DDL), mais especificamente o CREATE, ALTER e DROP.

Primeiro vamos ver como utilizar os comandos da DDL, iniciando pelo CREATE que serve para criar as tabelas no banco de dados, sendo a sua sintaxe muito simples.

```
CREATE TABLE nome_da_tabela ( ... );
```

Primeiro você define o comando, especifica que é uma tabela que está sendo criada e então o nome dessa tabela, logo em seguida cria o corpo da instrução para a definição das colunas. Veremos a seguir um exemplo bem simples da criação de uma tabela de clientes.

```
CREATE TABLE clientes (
   cpf VARCHAR(11) NOT NULL,
   nome VARCHAR(5) NOT NULL,
   dta_nascimento DATE NOT NULL,
   PRIMARY KEY (cpf)
);
```

O banco de dados Oracle possui diversos tipos de dados, por isso é interessante consultar a documentação para conhecer alguns deles com maior profundidade.

Agora que aprendemos a criar uma tabela e suas colunas, podemos alterar a estrutura dessa tabela ao utilizar o comando ALTER. Primeiro vamos criar outra tabela, a de notas fiscais.

```
CREATE TABLE notas_fisca(
     cpf VARCHAR(11) NOT NULL,
     data_venda DATE NOT NULL,
     imposto FLOAT NOT NULL
);
```

Nessa tabela de notas fiscais acabamos errando o seu nome, então vamos usar o ALTER TABLE para alterar o nome dessa tabela e corrigir o erro.

```
ALTER TABLE notas_fisca RENAME TO notas_fiscais;
```

Com essa alteração realizada agora temos o nome correto para esta tabela, mas observe que ela possui o campo CPF que é uma chave primaria da tabela clientes. Então vamos fazer o relacionamento entre essas tabelas, adicionando uma constrição de chave estrangeira para a tabela de notas fiscais.

```
ALTER TABLE notas_fiscais
```

```
ADD CONSTRAINT fk_clientes
FOREIGN KEY (cpf) REFERENCES clientes (cpf);
```

Com essa alteração dizemos que o cliente se relaciona com notas fiscais. Agora observe que na tabela de clientes só é aceito no campo nome até 5 caracteres, para modificarmos isso primeiro temos que especificar a tabela que vamos fazer a alteração, depois a coluna e finalmente a modificação que queremos realizar.

```
ALTER TABLE clientes
MODIFY nome
VARCHAR(30)
```

Pronto, agora a coluna nome da tabela de clientes pode aceitar valores maiores. E lembre-se sempre de consultar a documentação, pois existe várias modificações que podem ser feitas, como tornar uma coluna visível ou invisível, permitir ou não valores nulos e entre outras.

Um dos comandos mais temidos pelo seu poder destrutivo é o DROP, pois com ele você pode apagar um banco de dados inteiro. Entretanto veremos como excluir uma única tabela.

Digamos que não queremos mais a tabela de notas fiscais no nosso banco, então podemos usar o comando DROP e excluí-la da base de dados.

```
DROP TABLE notas_fiscais;
```

Simples assim, acabou de excluir uma tabela com milhões de dados.

Com a estrutura criada, alterada e até excluída, podemos começar a inserir e alterar os nossos dados, por isso veremos agora os comandos pertinentes a DML, como o INSERT, UPDATE e DELETE.

O comando INSERT permite fazer a inserção de dados na tabela, sendo uma linha por vez ou várias linhas em um “único” INSERT.

```
INSERT INTO nome_da_tabela VALUES (...valores...)
```

A sua sintaxe é bem simples, ela diz basicamente o seguinte “Insira nesta tabela os seguintes valores”. É importante dar atenção a ordem dos valores, pois eles devem estar na mesma ordem que as colunas da tabela.

```
INSERT INTO clientes
VALUES (‘587641587310’, ‘Paulo Henrique’, ‘12/09/1987’)
```

Agora vamos ver como inserir várias linhas de uma única vez em um banco de dados Oracle.

```
INSERT ALL
    INTO clientes VALUES ()
    INTO clientes VALUES ()
    ...
SELECT * FROM DUAL;
```

Observe que diferentemente de outros bancos de dados como o Postgresql, é necessário usar a cláusula ALL após o INSERT, e no final da *query* informar um SELECT, mesmo sendo da tabela DUAL.

O cliente que acabamos de inserir não nasceu em 1987, mas sim em 1997. Para corrigirmos o erro podemos usar o UPDATE e atualizar os dados que inserimos.

```
UPDATE clientes
SET dta_nascimento = ‘12/09/1997’
WHERE cpf = ‘587641587310’
```

A *query* diz “Atualize na tabela de clientes a coluna data de nascimento, inserindo nela a data x para todos os clientes que tenham este cpf”, é muito importante especificar uma condição para a atualizar os dados, pois se isso não for feito todas as linhas dessa coluna serão atualizadas para o novo valor.

O cliente que inserimos e depois atualizamos não deveria ter sido cadastrado. Para exclui-lo na nossa base de dados, podemos usar o comando DELETE, com muito cuidado, pois ele pode excluir todos os dados da tabela.

```
DELETE FROM clientes WHERE cpf = '587641587310';
```

Essa *query* exclui apenas o cliente que tem o cpf indicado, caso não tivéssemos especificado essa condição, todos os clientes teriam sido apagados, como na *query* a seguir.

```
DELETE FROM clientes;
```

Antes de finalizar essa seção, não entrarei em muitos detalhes, mas é importante entender que, todas as alterações realizadas, como alteração e exclusão, não são de fato efetivadas no banco de dados até que você use o COMMIT para efetivar as alterações, o que lhe permite usar o ROLLBACK caso tenha feito alguma mudança indevida. Esse mecanismo de COMMIT e ROLLBACK é um padrão do banco de dados, mas pode ser alterado pelo DBA, sendo assim é sempre muito importante se atentar ao que está fazendo, inclusive aos pequenos detalhes, evitando cometer erros significativos (como apagar mais de dois milhões de registros da tabela de folha de pagamentos).

2 PL/SQL

2.1 *Triggers*

Começaremos este capítulo falando sobre um tipo de bloco PL/SQL que é executado automaticamente quando ocorre alguma ação no banco de dados, as *triggers* ou gatilhos no português.

As *triggers* são blocos de comandos que podem ser executados quando ocorre alguma ação no banco de dados, como o INSERT, UPDATE ou DELETE, e esse bloco de comando pode ser executado antes (BEFORE) ou depois (AFTER) da ação realizada. Então você pode usar a *trigger* para fazer alguma coisa em uma tabela, mediante uma ação ocorrida em outra tabela, como por exemplo o log de usuários ou registros criados.

```
CREATE [OR REPLACE] TRIGGER nome_da_trigger
{BEFORE | AFTER} eventos ON nome_da_tabela
[FOR EACH ROW]
DECLARE
    declarações
BEGIN
    acoes_executadas
END;
```

A sintaxe da *trigger* diz o seguinte “Crie ou atualize a *trigger* x que vai ser executada quando acontecer alguma coisa na tabela y”. Essa não é a sintaxe completa, visto que não é o objetivo deste artigo se aprofundar em todos os aspectos da SQL e do banco de dados Oracle.

Como exemplo vamos criar duas tabelas, uma cliente e outra de log de clientes:

```
CREATE TABLE clientes
(
cpf VARCHAR(11) NOT NULL,
nome VARCHAR(30) NOT NULL,
dta_nascimento DATE NOT NULL,
idade INT NOT NULL,
PRIMARY KEY (cpf)
);

CREATE TABLE log_clientes
(
id_log NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY,
tabela VARCHAR(30) NOT NULL,
acao VARCHAR(6) NOT NULL,
usuario VARCHAR(30) NOT NULL,
data_log DATE NOT NULL,
```

```
PRIMARY KEY(id_log)
);
```

A tabela log de clientes ficará responsável por armazenar as ações feitas na tabela de clientes. Essas ações serão responsáveis por disparar a *trigger* que vai fazer o registro de log.

```
CREATE OR REPLACE TRIGGER tg_log_clientes
AFTER
INSERT OR UPDATE OR DELETE
ON clientes
FOR EACH ROW
DECLARE
    acao VARCHAR(6);
BEGIN
    acao := CASE
        WHEN INSERTING THEN 'INSERT'
        WHEN UPDATING THEN 'UPDATE'
        WHEN DELETING THEN 'DELETE'
END;
    INSERT INTO log_clientes (tabela, acao, usuario,
    data_log)
    VALUES('CLIENTES', acao, USER, SYSDATE);
END;
```

Agora quando inserir, atualizar ou excluir algum registro da tabela clientes, essa ação será refletida na *trigger*, salvado o nome da tabela onde foi realizada essa ação, a ação, o usuário e a data da ação.

Este foi apenas um exemplo da infinidade de procedimentos que podem ser realizados com *triggers*, podendo ser escrito um capítulo exclusivamente sobre isso. Entretanto, visando ser apenas uma introdução daremos continuidade no aprendizado da PL/SQL do banco de dados Oracle.