Apostila sobre o
Banco de Dados Postgre

Autor: Vanessa Rocha Solgate

Ano 2005

Introdução

O PostgreSQL é um sistema de gerenciamento de banco de dados objeto-relacional (SGBDOR) , ele foi o pioneiro em muitos conceitos objeto-relationais que agora estão se tornando disponíveis em alguns bancos de dados comerciais.

Desenvolvido no Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley. O projeto POSTGRES, liderado pelo Professor Michael Stonebraker, foi patrocinado pelas seguintes instituições: Defense Advanced Research Projects Agency (DARPA); Army Research Office (ARO); National Science Foundation (NSF); e ESL, Inc.

O PostgreSQL descende deste código original de Berkeley, possuindo o código fonte aberto. Fornece suporte às linguagens SQL92/SQL99, além de outras funcionalidades modernas.

Os Sistemas de Gerenciamento de Bancos de Dados Relacionais (SGBDR) tradicionais suportam um modelo de dados que consiste em uma coleção de relações com nome, contendo atributos de um tipo específico. Nos sistemas comerciais em uso, os tipos possíveis incluem número de ponto flutuante, inteiro, cadeia de caracteres, monetário e data.

Nesta apostila iremos aprender a como criar um banco de dados e sobre DDL (Data Definition Language) – Linguagem que os objetos que irão compor o banco de dados (comandos de criação e atualização da estrutura dos campos da tabela, por exemplo) e DML (Data Manipulation Language) - Linguagem que define os comandos de manipulação e operação dos dados (comandos de consulta e atualização dos dados das tabelas).

Parte A: Básico

1. Convenções para a Linguagem SQL

Convenção nada mais é que um padrão que iremos utilizar para criar os nossos objetos. A seguir segue uma tabela com as convenções mais utilizadas para o padrão da Linguagem SQL.

Convenção	Exemplo
Letra maiúscula	Comandos: SELECT, WHERE, AND, OR, ORDER BY, HAVING, CREATE, ALTER, INSERT, UPDATE, DELETE, etc.
Letra minúscula	Nome de tabelas e colunas, nome de funções, etc.

2. Criar Banco de Dados

Representa o arquivo físico de dados, armazenado em dispositivos periféricos, onde estão armazenados os dados de diversos sistemas, para consulta e atualização pelo usuário.
No nosso treinamento coloque o seu nome como Database.

Exemplo:

```
CREATE DATABASE vanessa
```

3. Criar um Schema

Um schema é uma coleção de objetos. Os objetos do schema são as estruturas lógicas que consultam diretamente aos dados em uma base de dados. Os objetos de um schema incluem tabelas, views, sinônimos, procedures, etc.

Exemplo:

```
CREATE SCHEMA sui;
```

Se uma tabela não pertence ao usuário, o nome do proprietário deve ser prefixado à tabela.

Exemplo:

```
SELECT * FROM sui.tsuitipoorgaocolegiado;
```

4. Tipos de Dados

Na tabela abaixo relacionaremos alguns tipos de Dados utilizados pelo Postgre:

4.1. Tipo Numérico

Nome do tipo	Tamanho de armazenamento
Smallint	2 bytes
Integer	4 bytes
Bigint	8 bytes
Decimal	variável
Numeric	variável
Real	4 bytes
double precision	8 bytes
Serial	4 bytes
Bigserial	8 bytes

Os tipos smallint, integer e bigint armazenam números inteiros, ou seja, números sem a parte fracionária, com diferentes faixas de valor. A tentativa de armazenar um valor fora da faixa permitida ocasiona erro.

O tipo integer é a escolha usual, porque oferece o melhor equilíbrio entre faixa de valores, tamanho de armazenamento e desempenho. Geralmente o tipo smallint só é utilizado quando o espaço em disco está muito escasso. O tipo bigint somente deve ser usado quando a faixa de valores de integer não for suficiente, porque este último é bem mais rápido.

O tipo bigint pode não funcionar de modo correto em todas as plataformas, porque depende de suporte no compilador para inteiros de oito bytes. Nas máquinas sem este suporte, o bigint age do mesmo modo que o integer (mas ainda demanda oito bytes para seu armazenamento). Entretanto, não é de nosso conhecimento nenhuma plataforma razoável onde este caso ainda se aplique.

O padrão SQL somente especifica os tipos inteiros integer (ou int) e smallint. O tipo bigint, e os nomes de tipo int2, int4 e int8 são extensões, também compartilhadas por vários outros sistemas de banco de dados SQL.

Os tipos de dado real e double precision são tipos numéricos de precisão variável não exatos. Na prática, estes tipos são geralmente implementações do padrão IEEE 754 para aritmética binária de ponto flutuante de precisão simples e dupla, respectivamente, conforme suportado pelo processador, sistema operacional e compilador utilizados.

4.2. Tipos para caracteres

Nome do tipo	Descrição
character varying(*n*), varchar(*n*)	comprimento variável com limite
character(*n*), char(*n*)	comprimento fixo, completado com brancos
text	comprimento variável não limitado

O SQL define dois tipos básicos para caracteres: character varying(*n*) e character(*n*), onde *n* é um número inteiro positivo. Estes dois tipos podem armazenar cadeias de caracteres com até *n* caracteres de comprimento. A tentativa de armazenar uma cadeia de caracteres mais longa em uma coluna de um destes tipos resulta em erro, a não ser que os caracteres excedentes sejam todos espaços. Neste caso a cadeia de caracteres será truncada em seu comprimento máximo (Esta exceção um tanto bizarra é requerida pelo padrão SQL). Se a cadeia de caracteres a ser armazenada for mais curta que o comprimento declarado, os valores do tipo character serão completados com espaço; os valores do tipo character varying simplesmente vão armazenar uma cadeia de caracteres mais curta.

As notações varchar(*n*) e char(*n*) são aliases para character varying(*n*) e character(*n*), respectivamente. O character sem especificação de comprimento é equivalente a character(1); se character varying for utilizado sem especificação de comprimento, este tipo aceita cadeias de caracteres de qualquer tamanho. Este último é uma extensão do PostgreSQL.

Além desses, o PostgreSQL suporta o tipo mais geral text, que armazena cadeias de caracteres de qualquer comprimento. Diferentemente de character varying, text não requer um limite superior explicitamente declarado de seu tamanho. Embora o tipo text não esteja no padrão SQL, muitos outros RDBMS também o incluem.

4.3. Tipos para data e hora

Tipo	Descrição	Armazenamento	Mais cedo	Mais tarde	Resolução
timestamp [(*p*)] [without time zone]	tanto data quanto hora	8 bytes	4713 AC	1465001 DC	1 microssegundo / 14 dígitos
timestamp [(*p*)] with time zone	tanto data quanto hora	8 bytes	4713 BC	AD 1465001	1 microssegundo / 14 dígitos
interval [(*p*)]	intervalos de tempo	12 bytes	-178000000 anos	178000000 anos	1 microssegundo
date	somente datas	4 bytes	4713 AC	32767 DC	1 dia
time [(*p*)] [without time zone]	somente a hora do dia	8 bytes	00:00:00.00	23:59:59.99	1 microssegundo
time [(*p*)] with time zone	somente a hora do dia	12 bytes	00:00:00.00+12	23:59:59.99-12	1 microssegundo

time, timestamp, e interval aceitam um valor opcional de precisão *p*, que especifica o número de dígitos fracionários presentes no campo de segundos. Por padrão não existe limite explícito para a precisão. O intervalo permitido para *p* é de 0 a 6 para os tipos timestamp e interval.

4.4. Tipo Booleano

O PostgreSQL disponibiliza o tipo boolean padrão do SQL. O tipo boolean pode possuir apenas dois estados: "verdade" ou "falso". O terceiro estado “desconhecido" é representado pelo valor nulo do SQL.

Os valores literais válidos para o estado "verdade" são:

TRUE
't'
'true'
'y'
'yes'
'1'

Para o estado "falso" os seguintes valores podem ser utilizados:

FALSE
'f'
'false'
'n'
'no'
'0'

A utilização das palavras chave *TRUE* e *FALSE* é preferida (e em conformidade com o padrão SQL).

OBS: Existem outros tipos de dados que não será possível relatar nesta apostila, você pode consultar o ***Guia do Usuário do PostgreSQL 7.3.4.***

5. Criar a estrutura de uma tabela

5.1. Regras de Nomeação

Nomes de tabelas e nomes de colunas:

Devem começar com uma letra

Devem conter somente A_Z, a_z, 0_9, _, $, e #

Não devem duplicar o nome de um outro objeto de um mesmo proprietário.

5.2. Como criar uma tabela

Uma tabela em um banco de dados relacional é muito semelhante a uma tabela no papel: é composta por linhas e colunas. O número e a ordem das colunas são fixos, e cada coluna possui um nome. O número de linhas é variável, refletindo a quantidade de dados armazenados em um determinado instante. O SQL não dá nenhuma garantia relativa à ordem das linhas na tabela. Quando uma tabela é lida, as linhas aparecem em uma ordem aleatória, a não ser que uma ordenação seja explicitamente requisitada.

Cada coluna possui um tipo de dado. O tipo de dado restringe o conjunto de valores que podem ser atribuídos à coluna.

Para criar uma tabela é utilizado o comando ***CREATE TABLE***, próprio para esta tarefa. Neste comando são especificados ao menos o nome da nova tabela, os nomes das colunas, e os tipos de dado de cada coluna.

Exemplo:

```
CREATE TABLE tsuitipoorgaocolegiado
(    numtipoorgaocol     INTEGER ,
     strtipoorgaocol     VARCHAR(200)
);
```

O tipo *numeric* pode armazenar a parte fracionária, usual em valores monetários.

5.3. Campos com restrição de não-nulo

Uma restrição de não-nulo simplesmente especifica que uma coluna não pode conter o valor nulo. Um exemplo da sintaxe:

Exemplo:

```
CREATE TABLE tsuitipoorgaocolegiado
(
     numtipoorgaocol     INTEGER NOT NULL,
     strtipoorgaocol     VARCHAR(200) NOT NULL
);
```

Uma restrição de não-nulo é sempre escrita como restrição de coluna.

5.4. Coluna com valor padrão

Uma coluna pode possuir um valor padrão. Quando uma nova linha é criada, e nenhum valor é especificado para algumas colunas, o valor padrão de cada uma destas colunas é atribuído à mesma.

Na definição da tabela, o valor padrão é posicionado após o tipo de dado da coluna. Por exemplo:

```
CREATE TABLE sui.tsuientidade
(
  numentidade int4 NOT NULL,
  chrnatent char(1) NOT NULL DEFAULT 'I',
  chrtpent char(1) NOT NULL DEFAULT 'U'
)
```

5.5. Restrição de unicidade

A restrição de unicidade garante que os dados contidos na coluna, ou no grupo de colunas, é único em relação a todas as outras linhas da tabela.

Exemplo:

```
CREATE TABLE tsuitipoorgaocolegiado
(    numtipoorgaocol     INTEGER NOT NULL,
     strtipoorgaocol     VARCHAR(200) NOT NULL UNIQUE
);
```

Quando escrita como restrição de coluna, e

```
CREATE TABLE tsuitipoorgaocolegiado
(    numtipoorgaocol     INTEGER NOT NULL,
     strtipoorgaocol     VARCHAR(200) NOT NULL,
     UNIQUE (strtipoorgaocol)
);
```

Quando escrita como restrição de tabela.

Também é possível atribuir nomes às restrições de unicidade:

```
CREATE TABLE tsuitipoorgaocolegiado
( numtipoorgaocol   INTEGER NOT NULL,
  strtipoorgaocol   VARCHAR(200) NOT NULL
     CONSTRAINT uk_strtipoorgaocol UNIQUE,
);
```

5.6. Restrição de verificação

Uma restrição de verificação é o tipo mais genérico de restrição. Permite especificar que os valores de uma determinada coluna devem estar de acordo com uma expressão arbitrária. Exemplo::

```
CREATE TABLE sui.tsuitipoorgaocolegiado
(
  numtipoorgaocol int4 NOT NULL,
  strtipoorgaocol varchar(200) NOT NULL,
  chrtipo char(1) NOT NULL CHECK (chrtipo = 'C' OR
                                  chrtipo = 'L')
);
```

Como pode ser observada, a definição da restrição está posicionada após o tipo de dado, do mesmo modo que a definição de valor padrão. O valor padrão e as restrições podem estar em qualquer ordem. Uma restrição de verificação é composta pela palavra chave ***CHECK*** seguida por uma expressão entre parênteses. A expressão de restrição de verificação deve envolver a coluna sendo restringida, senão não fará muito sentido.

Também pode ser atribuído um nome individual para a restrição. Isto torna mais clara a mensagem de erro, e permite fazer referência à restrição quando for desejado alterá-la. A sintaxe é:

```
CREATE TABLE sui.tsuitipoorgaocolegiado
(
  numtipoorgaocol int4 NOT NULL,
  strtipoorgaocol varchar(200) NOT NULL,
  chrtipo char(1) NOT NULL
  CONSTRAINT ck_tsuitipoorgaocolegiado_chrtipo CHECK
     (chrtipo = 'C' OR chrtipo = 'L')
);
```

5.7. Chave Primária

A chave primária indica que a coluna, ou grupo de colunas pode ser utilizado como identificador único para as linhas da tabela.

Somente uma chave primária pode ser especificada para uma tabela, seja como uma restrição de coluna ou como uma restrição de tabela.

A restrição de chave primária deve abranger um conjunto de colunas que seja diferente de outro conjunto de colunas abrangido por uma restrição de unicidade definida para a mesma tabela.

A teoria de banco de dados relacional determina que toda tabela deve ter uma chave primária.

Exemplo:

```
CREATE TABLE tsuitipoorgaocolegiado
(    numtipoorgaocol     INTEGER NOT NULL PRIMARY KEY,
     strtipoorgaocol     VARCHAR(200) NOT NULL
);
```

Você poderá criar uma chave primária com duas ou mais colunas:

```
CREATE TABLE exemplo (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

5.8. Chave estrangeira

A restrição de chave estrangeira especifica que o valor da coluna (ou grupo de colunas) deve corresponder a algum valor que existe em uma linha de outra tabela. Diz-se que este comportamento mantém a *integridade referencial* entre duas tabelas relacionadas.

Supondo que já temos a tabela de Tipo de Órgão Colegiado utilizada diversas vezes anteriormente:

```
CREATE TABLE tsuitipoorgaocolegiado
(    numtipoorgaocol     INTEGER NOT NULL PRIMARY KEY,
     strtipoorgaocol     VARCHAR(200) NOT NULL
);
```

Agora vamos supor, também, que existe uma tabela armazenando os Órgãos Colegiados destes Tipos de Órgãos Colegiados, e desejamos garantir que a tabela de Órgãos Colegiados somente contenha tipos de órgão de colegiado que realmente existem. Para isso é definida uma restrição de chave estrangeira na tabela Órgão Colegiado, fazendo referência à tabela Tipo de Órgão Colegiado.

Exemplo:

```
CREATE TABLE tsuiorgaocolegiado
( numorgaocol INTEGER NOT NULL,
  tipoorgaocoleg_numtipoorg INTEGER NOT NULL REFERENCES
  tsuitipoorgaocolegiado (numtipoorgaocol),
  strnomeorgaocol VARCHAR(100) NOT NULL,
  datativorgaocol DATE NOT NULL,
  strsiglaorgaocol VARCHAR(10) ,
  datdesativorgaocol DATE
  strfunlegalorgaocol VARCHAR(200),
  strstatus CHAR(1) NOT NULL
);
```

Isto torna impossível criar Órgaos Colegiados com ocorrências de tipoorgaocoleg_numtipoorg que não existam na tabela Tipo de Órgão Colegiado.

Nesta situação é dito que a tabela Órgaos Colegiados é a tabela *que faz referência*, e a tabela Tipo de Órgão Colegiado é a tabela *referenciada*. Da mesma forma existem colunas fazendo referência e sendo referenciadas.

O comando acima pode ser abreviado escrevendo-se:

```
CREATE TABLE tsuiorgaocolegiado
( numorgaocol INTEGER NOT NULL,
  tipoorgaocoleg_numtipoorg INTEGER NOT NULL
    REFERENCES tsuitipoorgaocolegiado,
  strnomeorgaocol VARCHAR(100) NOT NULL,
  datativorgaocol DATE NOT NULL,
  strsiglaorgaocol VARCHAR(10) ,
  datdesativorgaocol DATE
  strfunlegalorgaocol VARCHAR(200),
  strstatus CHAR(1) NOT NULL
);
```

Na ausência da lista de colunas, a chave primária da tabela referenciada é assumida como sendo a coluna referenciada.

Obviamente, o número e o tipo das colunas na restrição precisam corresponder ao número e tipo das colunas referenciadas.

6. Alterando a estrutura de uma tabela

Quando percebemos, após a tabela ter sido criada, que foi cometido um erro ou que as necessidades da aplicação mudaram, é possível remover a tabela e criá-la novamente. Porém, este procedimento não é conveniente quando existem dados na tabela, ou se a tabela é referenciada por outros objetos do banco de dados (por exemplo, uma restrição de chave estrangeira). Para esta finalidade o PostgreSQL disponibiliza um conjunto de comandos que realizam modificações em tabelas existentes.

Pode ser feito:

Incluir coluna;

Excluir coluna;

Incluir restrição;

Excluir restrição;

Todas estas atividades são realizadas utilizando o comando *ALTER TABLE*.

6.1. Adicionando novas colunas

Para incluir uma coluna deve ser utilizado o comando:

```
ALTER TABLE tsuitipoorgaocolegiado ADD COLUMN chrtipo char(1);
```

Inicialmente a nova coluna conterá valores nulos nas linhas existentes na tabela.

6.2. Removendo coluna da tabela

Para excluir uma coluna deve ser utilizado o comando:

```
ALTER TABLE tsuitipoorgaocolegiado DROP COLUMN chrtipo;
```

6.3. Adicionando restrição

É utilizada a sintaxe de restrição de tabela para incluir uma nova restrição.

Por exemplo:

```
ALTER TABLE tsuitipoorgaocolegiado
ADD CONSTRAINT uk_strtipoorgaocol UNIQUE (strtipoorgaocol);

ALTER TABLE tsuiorgaocolegiado
ADD FOREIGN KEY (tipoorgaocoleg_numtipoorg)
REFERENCES tsuitipoograocolegiado;
```

Para adicionar uma restrição de não nulo, que não pode ser escrita na forma de restrição de tabela, deve ser utilizada a sintaxe:

```
ALTER TABLE tsuitipoorgaocolegiado
ALTER COLUMN chrtipo SET NOT NULL;
```

A restrição será verificada imediatamente, portanto os dados da tabela devem satisfazer a restrição antes desta ser criada.

6.4. Removendo restrição

Para excluir uma restrição é necessário conhecer seu nome. Quando o usuário atribuiu um nome à restrição é fácil, caso contrário o sistema atribui para a restrição um nome gerado que precisa ser descoberto. O comando *\d nome_da_tabela* do psql pode ser útil nesta situação; outras interfaces também podem oferecer um modo de inspecionar os detalhes das tabelas. O comando a ser utilizado para excluir a restrição é:

```
ALTER TABLE tsuitipoorgaocolegiado
DROP CONSTRAINT uk_strtipoorgaocol;
```

Esta sintaxe serve para todos os tipos de restrição exceto não-nulo. Para excluir uma restrição de não-nulo deve ser utilizado o comando:

```
ALTER TABLE tsuitipoorgaocolegiado
ALTER COLUMN chrtipo DROP NOT NULL;
```

(Lembre-se que as restrições de não-nulo não possuem nome).

7. Excluindo uma tabela

Finalmente, deve ser mencionado que se a tabela não for mais necessária, ou se deseja recriá-la de uma forma diferente, esta pode ser removida através do seguinte comando:

```
DROP TABLE tsuitipoorgaocolegiado;
```

8. Inclusão de dados em uma tabela

Ao ser criada a tabela não contém nenhum dado. A primeira coisa a ser feita para o banco de dados ser útil é colocar dados. Conceitualmente, os dados são inseridos uma linha por vez. É claro que pode ser inserida mais de uma linha, mas não existe modo de inserir menos de uma linha de cada vez. Mesmo conhecendo apenas o valor de algumas colunas, uma linha inteira deve ser criada.

Para criar uma nova linha deve ser utilizado o comando *INSERT*.

Um comando mostrando a inclusão de uma linha pode ser:

```
INSERT INTO tsuitipoorgaocolegiado
VALUES (1, 'CONSELHOS DE DEPARTAMENTO', ‘L’);
```

Os valores dos dados são colocados na mesma ordem que as colunas aparecem na tabela, separados por vírgula. Geralmente os valores dos dados são literais (constantes), mas expressões escalares também são permitidas.

A sintaxe mostrada acima tem como desvantagem necessitar o conhecimento da ordem das colunas da tabela. Para evitar isto, as colunas podem ser declaradas explicitamente. Por exemplo, os dois comandos mostrados abaixo possuem o mesmo efeito do comando mostrado acima:

```
INSERT INTO tsuitipoorgaocolegiado
(numtipoorgaocol, strtipoorgaocol, chrtipo)
VALUES (1, 'CONSELHOS DE DEPARTAMENTO', ‘L’);

INSERT INTO tsuitipoorgaocolegiado
(strtipoorgaocol, chrtipo, numtipoorgaocol)
VALUES ('CONSELHOS DE DEPARTAMENTO', ‘L’, 1);
```

Muitos usuários consideram boa prática escrever sempre os nomes das colunas.

9. Consultando dados da tabela

O processo de trazer de volta, ou o comando para trazer os dados armazenados no banco de dados, é chamado de *consulta*. No SQL, o comando SELECT é utilizado para especificar as consultas. A sintaxe geral do comando SELECT é:

```
SELECT lista_seleção
FROM expressão_tabela [especificação_ordenação]
```

As próximas seções descrevem os detalhes da lista de seleção, da expressão de tabela, e da especificação da ordenação.

O tipo mais simples de consulta possui a forma:

```
SELECT * FROM tsuitipoorgaocolegiado;
```

Este comando traz todas as linhas e todas as colunas da *tsuitipoorgaocolegiado*. A forma de trazer depende da aplicação cliente. Por exemplo, o programa psql exibe uma tabela ASCII na tela, enquanto as bibliotecas cliente possuem funções para obter linhas e colunas individualmente. O * especificado na lista de seleção significa todas as colunas que a expressão de tabela tem para oferecer. A lista de seleção também pode especificar um subconjunto das colunas disponíveis, ou efetuar cálculos utilizando as colunas. Por exemplo, se *tsuitipoorgaocolegiado* possui colunas chamadas *numtipoorgaocol*, *strtipoorgaocol* e chrtipo, pode ser feita a seguinte consulta:

```
SELECT numtipoorgaocol, strtipoorgaocol, chrtipo
FROM tsuitipoorgaocolegiado;
```

9.1. Alias de coluna

Podem ser atribuídos nomes para as entradas da lista de seleção para processamento posterior. Neste caso "processamento posterior" é uma especificação opcional de ordenação e a aplicação cliente (por exemplo, os títulos das colunas para exibição). Por exemplo:

```
SELECT numtipoorgaocol AS Codigo,
strtipoorgaocol AS Descricao
FROM...
```

Se nenhum nome de coluna de saída for especificado utilizando AS, o sistema atribui um nome padrão. Para referências simples a colunas é o nome da coluna referenciada.

9.2. Filtrando os dados

A sintaxe da cláusula *WHERE* é:

WHERE condição_pesquisa

onde a *condição_pesquisa* é qualquer expressão de valor.

Abaixo estão mostrados alguns exemplos da cláusula *WHERE*:

```
SELECT * FROM tsuitipoorgaocolegiado
WHERE numtipoorgaocol > 5

SELECT * FROM tsuitipoorgaocolegiado
WHERE numtipoorgaocol IN (1, 2, 3)

SELECT * FROM tsuitipoorgaocolegiado
WHERE numtipoorgaocol IN
     (SELECT tipoorgaocoleg_numtipoorg
      FROM tsuiorgaocolegiado)

SELECT * FROM tsuitipoorgaocolegiado
WHERE numtipoorgaocol BETWEEN 5 AND 8
```

9.3. Removendo linhas duplicadas na consulta

Após a lista de seleção ser processada, a tabela resultante pode opcionalmente estar sujeita a remoção das linhas duplicadas. A palavra chave *DISTINCT* deve ser escrita logo após o *SELECT* para ativar esta funcionalidade:

```
SELECT DISTINCT strtipoorgaocol
FROM tsuitipoorgaocolegiado
```

Como é óbvio, duas linhas são consideradas distintas quando tiverem pelo menos uma coluna diferente. Os valores nulos são considerados iguais nesta comparação.

9.4. Ordenação de linhas

Após a consulta ter produzido uma tabela de saída (após a lista de seleção ter sido processada) esta tabela pode, opcionalmente, ser ordenada. Se nenhuma ordenação for especificada, as linhas retornam em uma ordem aleatória. Na verdade, neste caso a ordem depende dos tipos de plano de varredura e de junção e da ordem no disco, mas não se deve confiar nisto. Uma determinada ordem de saída somente pode ser garantida se a etapa de ordenação for explicitamente especificada.

A cláusula *ORDER BY* especifica a ordenação:

```
SELECT lista_seleção
 FROM expressão_tabela
 ORDER BY coluna1 [ASC | DESC] [, coluna2 [ASC | DESC] ...]
```

onde *coluna1*, etc. fazem referência às colunas da lista de seleção. Pode ser tanto o nome de saída de uma coluna quanto o número da coluna.

Alguns exemplos:

```
SELECT a, b FROM tabela1 ORDER BY a;
SELECT a + b AS soma, c FROM tabela1 ORDER BY soma;
SELECT a, sum(b) FROM tabela1 GROUP BY a ORDER BY 1;
```

Como uma extensão do padrão SQL, o PostgreSQL também permite a ordenação por expressões arbitrárias:

```
SELECT a, b FROM tabela1 ORDER BY a + b;
```

Também é permitido fazer referência a nomes de colunas da cláusula *FROM* que foram renomeados na lista de seleção:

```
SELECT chrtipo AS tipo
FROM tsuitipoorgaocolegiado
ORDER BY tipo;
```

Mas estas extensões não funcionam nas consultas que envolvem *UNION*, *INTERSECT* ou *EXCEPT*, e não são portáveis para outros bancos de dados SQL.

Cada coluna especificada pode ser seguida pela palavra opcional *ASC* ou *DESC*, para determinar a forma de ordenação como ascendente ou descendente. A forma *ASC* é o padrão. A ordenação ascendente coloca os valores menores na frente, sendo que "menor" é definido nos termos do operador <. De forma semelhante, a ordenação descendente é determinada pelo operador >.

Se mais de uma coluna de ordenação for especificada, as últimas colunas são utilizadas para ordenar as linhas iguais na ordem imposta pelas primeiras colunas ordenadas.

9.5. Operadores lógicos

Os operadores lógicos habituais estão disponíveis:

AND

OR

NOT

O SQL utiliza a lógica booleana de três valores, onde o valor nulo representa o "desconhecido". Observe as seguintes tabelas verdade:

a	b	*a* AND *b*	*a* OR *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Esses operadores booleanos arbitrários (AND, OR e NOT) são permitidos na qualificação da consulta.

Exemplo:

```
SELECT *
FROM tsuitipoorgaocolegiado
WHERE chrtipo = ‘L’
AND numtipoorgaocol > 5;
```

9.6. Operadores de comparação

Os operadores de comparação habituais são:

Operador	Descrição
<	menor que
>	maior que
<=	menor que ou igual a
>=	maior que ou igual a
=	igual
<> ou *!=*	diferente

Os operadores de comparação estão disponíveis em todos os tipos de dado onde fazem sentido. Todos os operadores de comparação são operadores binários que retornam valores do tipo boolean; expressões como *1 < 2 < 3* não são válidas (porque não existe o operador < para comparar um valor booleano com *3*).

Além dos operadores de comparação, a construção especial BETWEEN está disponível.

a BETWEEN *x* AND *y*

equivale a

a >= *x* AND *a* <= *y*

Igualmente,

a NOT BETWEEN *x* AND *y*

equivale a

a < *x* OR *a* > *y*

Não existe diferença entre estas duas formas, além dos ciclos de CPU necessários para reescrever a primeira forma na segunda internamente.

Para verificar se um valor é ou não nulo devem ser usadas as construções:

expressão IS NULL

expressão IS NOT NULL

9.7. Funções e operadores matemáticos

Estão disponíveis operadores matemáticos para vários tipos de dado do PostgreSQL. Para os tipos sem as convenções matemáticas habituais para todas as permutações possíveis (por exemplo, os tipos de data e hora).

Nome	Descrição	Exemplo	Resultado
+	adição	2 + 3	5
-	subtração	2 - 3	-1
*	multiplicação	2 * 3	6
/	divisão (divisão inteira trunca o resultado)	4 / 2	2
%	módulo (resto)	5 % 4	1
^	exponenciação	2.0 ^ 3.0	8

Nome	Descrição	Exemplo	Resultado
|/	raiz quadrada	|/ 25.0	5
||/	raiz cúbica	||/ 27.0	3
!	fatorial	5 !	120
!!	fatorial (operador de prefixo)	!! 5	120
@	valor absoluto	@ -5.0	5
&	AND binário	91 & 15	11
|	OR binário	32 | 3	35
#	XOR binário	17 # 5	20
~	NOT binário	~1	-2
<<	deslocamento binário à esquerda	1 << 4	16
>>	deslocamento binário à direita	8 >> 2	2

9.8. Funções e operadores para cadeia de caracteres

Esta seção descreve as funções e operadores disponíveis para examinar e manipular cadeias de caracteres. Neste contexto as cadeias de caracteres incluem valores dos tipos `CHARACTER`, `CHARACTER VARYING` e `TEXT`. A menos que seja dito o contrário, todas as funções listadas abaixo trabalham com todos estes tipos, mas deve ser tomado cuidado quando for utilizado o tipo `CHARACTER` com os efeitos em potencial do preenchimento automático. De modo geral, as funções descritas nesta seção também trabalham com dados que não são cadeias de caracteres, convertendo estes dados primeiro na representação de cadeia de caracteres. Algumas funções também existem em forma nativa para os tipos cadeia de bits.

Função	Tipo retornado	Descrição	Exemplo	Resultado
cadeia_de_caracteres || cadeia_de_caracteres	text	Concatenação de cadeias de caracteres	'Post' || 'greSQL'	PostgreSQL
char_length(cadeia_de_caracteres) ou character_length(cadeia_de_caracteres)	integer	Número de caracteres na cadeia de caracteres	char_length('jose')	4
lower(cadeia_de_caracteres)	text	Converte a cadeia de caracteres em letras minúsculas	lower('TOM')	tom
position(caracteres in cadeia_de_caracteres)	integer	Localização dos caracteres	position('om' in 'Thomas')	3

Função	Tipo retornado	Descrição	Exemplo	Resultado
		especificados		
substring(*cadeia_de_caracteres* [from integer] [for integer])	text	Extrai parte da cadeia de caracteres	substring('Thomas' from 2 for 3)	hom
substring(cadeia_de_caracteres from expressão)	text	Extrai a parte da cadeia de caracteres correspondente à expressão regular POSIX	substring('Thomas' from '...$')	mas
substring(cadeia_de_caracteres from expressão for escape)	text	Extrai a parte da cadeia de caracteres correspondente à expressão regular SQL	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [*caracteres*] from *cadeia_de_caracteres*)	text	Remove da extremidade inicial/final/ambas da *cadeia_de_caracteres*, a cadeia de caracteres mais longa contendo apenas os *caracteres* (espaço por padrão)	trim(both 'x' from 'xTomxx')	Tom
upper(cadeia_de_caracteres)	text	Converte a cadeia de caracteres em letras maiúsculas	upper('tom')	TOM
ascii(text)	integer	código ASCII do primeiro caractere do argumento	ascii('x')	120
btrim(cadeia_de_caracteres text, trim text)	text	Remove (trim) a maior cadeia de caracteres composta apenas pelos caracteres contidos em *trim*, do início e do fim da *cadeia_de_caracteres*	btrim('xyxtrimyyx','xy')	trim
initcap(text)	text	Converte a primeira letra de cada palavra (separadas por espaço em branco) em maiúscula	initcap('hi thomas')	Hi Thomas
length(cadeia_de_caracteres)	integer	Comprimento da cadeia de caracteres	length('jose')	4

Função	Tipo retornado	Descrição	Exemplo	Resultado
lpad(cadeia_de_caracteres text, comprimento integer [, preenchimento text])	text	Preenche a *cadeia_de_caracteres* até o *comprimento* adicionando os caracteres de *preenchimento* (espaço por padrão). Se a *cadeia_de_caracteres* for mais longa que o *comprimento* então é truncada (à direita).	lpad('hi', 5, 'xy')	xyxhi
ltrim(cadeia_de_caracteres text, text text)	text	Remove do início da cadeia de caracteres, a cadeia de caracteres mais longa contendo apenas caracteres de *trim*	ltrim('zzzytrim','xyz')	trim
replace(cadeia_de_caracteres text, origem text, destino text)	text	Substitui todas as ocorrências de *origem* na *cadeia_de_caracteres* por *destino*	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
rpad(cadeia_de_caracteres text, comprimento integer [, preenchimento text])	text	Preenche a *cadeia_de_caracteres* até o *comprimento* adicionando os caracteres de *preenchimento* (espaço por padrão). Se a *cadeia_de_caracteres* for mais longa que o *comprimento* estão é truncada.	rpad('hi', 5, 'xy')	hixyx
rtrim(cadeia_de_caracteres text, trim text)	text	Remove do fim da cadeia de caracteres, a cadeia de caracteres mais longa contendo apenas caracteres de *trim*	rtrim('trimxxxx','x')	trim
substr(cadeia_de_caracteres, origem [, contador])	text	Extrai a substring especificada (o mesmo que substring(cadeia_de	substr('alphabet', 3, 2)	ph

Função	Tipo retornado	Descrição	Exemplo	Resultado
		caracteres from origem for contador))		
translate(cadeia_de_caracteres text, origem text, destino text)	text	Todo caractere da *cadeia_de_caracteres*, correspondente a um caractere do conjunto *origem*, é substituído pelo caractere correspondente do conjunto *destino*.	translate('12345', '14', 'ax')	a23x5

9.9. Usando a condição LIKE

Use a condição LIKE para executar buscas de valores válidos de uma string.

As condições da busca podem conter caracteres literais ou números:

% representa qualquer seqüência de zero ou mais caracteres

_ representa um único caractere.

Exemplos:

```
SELECT strtipoorgaocol
FROM tsuitipoorgaocolegiado
WHERE strtipoorgaocol LIKE 'CON%’;

SELECT strtipoorgaocol
FROM tsuitipoorgaocolegiado
WHERE strtipoorgaocol LIKE ‘_O%’;
```

9.10. Funções para formatar tipos de dados

As funções de formatação do PostgreSQL disponibilizam um poderoso conjunto de ferramentas para converter diversos tipos de dado (*date/time, integer, floating point, numeric*) em cadeias de caracteres formatadas, e para converter cadeias de caracteres formatadas nos tipos de dado especificados.

Funções de formatação

Função	Retorna	Descrição	Exemplo
to_char(timestamp, text)	Text	converte carimbo de tempo (*time stamp*) em cadeia de caracteres	to_char(timestamp 'now','HH12:MI:SS')
to_char(interval, text)	Text	converte intervalo em cadeia de caracteres	to_char(interval '15h 2m 12s','HH24:MI:SS')
to_char(int, text)	Text	converte inteiro em cadeia de caracteres	to_char(125, '999')
to_char(double precision, text)	Text	converte real e precisão dupla em cadeia de caracteres	to_char(125.8, '999D9')
to_char(numeric, text)	Text	converte numérico em cadeia de caracteres	to_char(numeric '-125.8', '999D99S')
to_date(text, text)	Date	converte cadeia de caracteres em data	to_date('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(text, text)	timestamp	converte cadeia de caracteres em carimbo de tempo	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	converte cadeia de caracteres em numérico	to_number('12,454.8-', '99G999D9S')

A seguir na tabela, elementos que podem ser utilizados na máscara para formatar valores de data e de hora.

Elementos para máscara de conversão de data e hora

Elemento	Descrição
HH	hora do dia (01-12)
HH12	hora do dia (01-12)
HH24	hora do dia (00-23)
MI	minuto (00-59)
SS	segundo (00-59)
MS	milissegundo (000-999)
US	microssegundo (000000-999999)
SSSS	segundos após a meia-noite (0-86399)
AM ou *A.M.* ou *PM*	indicador de meridiano (maiúsculas)

Elemento	Descrição
ou *P.M.*	
am ou *a.m.* ou *pm* ou *p.m.*	indicador de meridiano (minúsculas)
Y,YYY	ano (4 e mais dígitos) com vírgula
YYYY	ano (4 e mais dígitos)
YYY	últimos 3 dígitos do ano
YY	últimos 2 dígitos do ano
Y	último dígito do ano
BC ou *B.C.* ou *AD* ou *A.D.*	indicador de era (maiúscula)
bc ou *b.c.* ou *ad* ou *a.d.*	indicador de era (minúscula)
MONTH	nome completo do mês em maiúsculas (9 caracteres completado com espaços)
Month	nome completo do mês em maiúsculas e minúsculas (9 caracteres completado com espaços)
month	nome completo do mês em minúsculas (9 caracteres completado com espaços)
MON	nome abreviado do mês em maiúsculas (3 caracteres)
Mon	nome abreviado do mês em maiúsculas e minúsculas (3 caracteres)
mon	nome abreviado do mês em minúsculas (3 caracteres)
MM	número do mês (01-12)
DAY	nome completo do dia em maiúsculas (9 caracteres completado com espaços)
Day	nome completo do dia em maiúsculas e minúsculas (9 caracteres completado com espaços)
day	nome completo do dia em minúsculas (9 caracteres completado com espaços)
DY	nome abreviado do dia em maiúsculas (3 caracteres)
Dy	nome abreviado do dia em maiúsculas e minúsculas (3 caracteres)
dy	nome abreviado do dia em minúsculas (3 caracteres)
DDD	dia do ano (001-366)
DD	dia do mês (01-31)
D	dia da semana (1-7; SUN=1)
W	semana do mês (1-5) onde a primeira semana começa no primeiro dia do mês
WW	número da semana do ano (1-53) onde a primeira semana começa no primeiro dia do ano

Elemento	Descrição
IW	número da semana do ano ISO (A primeira quinta-feira do novo ano está na semana 1)
CC	século (2 dígitos)
J	Dia Juliano (dias desde 1 de janeiro de 4712 AC)
Q	trimestre
RM	mês em algarismos romanos (I-XII; I=Janeiro) - maiúsculas
rm	mês em algarismos romanos (I-XII; I=Janeiro) - minúsculas
TZ	zona horária - maiúsculas
tz	zona horária - minúsculas

Certos modificadores podem ser aplicados a qualquer elemento da máscara para alterar seu comportamento. Por exemplo, "*FMMonth*" é o elemento "*Month*" com o prefixo "*FM*".

Modificadores dos elementos das máscara de conversão de data e hora

Modificador	Descrição	Exemplo
prefixo *FM*	modo de preenchimento (suprime completar com brancos e zeros)	FMMonth
sufixo *TH*	adicionar o sufixo de número ordinal em maiúsculas	DDTH
sufixo *th*	adicionar o sufixo de número ordinal em minúsculas	DDth
prefixo *FX*	opção global de formato fixo (veja nota de utilização)	FX Month DD Day
sufixo *SP*	modo de falar (*spell mode*) (ainda não implementado)	DDSP

Elementos para máscara de conversão numérica

Elemento	Descrição
9	valor com o número especificado de dígitos
0	valor com zeros à esquerda
. (ponto)	ponto decimal
, (vírgula)	separador de grupo (milhares)
PR	valor negativo entre chaves
S	valor negativo com o sinal de menos (utiliza a localização)
L	símbolo da moeda (utiliza a localização)
D	ponto decimal (utiliza a localização)
G	separador de grupo (utiliza a localização)

Elemento	Descrição
MI	sinal de menos na posição especificada (se número < 0)
PL	sinal de mais na posição especificada (se número > 0)
SG	sinal de mais/menos na posição especificada
RN	algarismos romanos (entrada entre 1 e 3999)
TH ou *th*	converte em número ordinal
V	desloca *n* dígitos (veja as notas)
EEEE	notação científica (ainda não implementada)

Exemplos da função to_char

Entrada	Saída
to_char(now(),'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(now(),'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1,'99.99')	' -.10'
to_char(-0.1,'FM9.99')	'-.1'
to_char(0.1,'0.9')	' 0.1'
to_char(12,'9990999.9')	' 0012.0'
to_char(12,'FM9990999.9')	'0012'
to_char(485,'999')	' 485'
to_char(-485,'999')	'-485'
to_char(485,'9 9 9')	' 4 8 5'
to_char(1485,'9,999')	' 1,485'
to_char(1485,'9G999')	' 1 485'
to_char(148.5,'999.999')	' 148.500'
to_char(148.5,'999D999')	' 148,500'
to_char(3148.5,'9G999D999')	' 3 148,500'
to_char(-485,'999S')	'485-'
to_char(-485,'999MI')	'485-'
to_char(485,'999MI')	'485'
to_char(485,'PL999')	'+485'
to_char(485,'SG999')	'+485'
to_char(-485,'SG999')	'-485'
to_char(-485,'9SG99')	'4-85'
to_char(-485,'999PR')	'<485>'
to_char(485,'L999')	'DM 485
to_char(485,'RN')	' CDLXXXV'
to_char(485,'FMRN')	'CDLXXXV'

Entrada	Saída
to_char(5.2,'FMRN')	V
to_char(482,'999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8,'"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12,'99V999')	' 12000'
to_char(12.4,'99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

10. Atualizando linhas da tabela

A modificação dos dados armazenados no banco de dados é referida como atualização. Pode ser atualizada uma linha da tabela, todas as linhas da tabela, ou um subconjunto das linhas. Cada coluna pode ser atualizada individualmente; as outras colunas não são afetadas.

Para efetuar uma atualização são necessárias três informações:

O nome da tabela e da coluna;

O novo valor para a coluna;

Quais linhas serão atualizadas.

Por exemplo, o comando mostrado abaixo atualiza o Tipo de Órgão Colegiado igual a 1 para o tipo “L” (Local):

```
UPDATE tsuitipoorgaocolegiado
SET chrtipo = ‘L’
WHERE numtipoorgaocol = 1;
```

Este comando faz nenhuma, uma, ou muitas linhas serem atualizadas. Não é errado tentar fazer uma atualização sem nenhuma linha correspondente.

Vejamos este comando em detalhe: Primeiro aparece a palavra chave *UPDATE* seguida pelo nome da tabela. Como usual, o nome da tabela pode ser qualificado pelo esquema, senão é procurado no caminho. Depois aparece a palavra chave *SET*, seguida pelo nome da coluna, por um sinal de igual, e do novo valor da coluna. O novo valor da coluna pode ser qualquer expressão escalar, e não apenas uma constante.

Pode ser deixada de fora a cláusula *WHERE*. Quando a cláusula *WHERE* é omitida significa que todas as linhas da tabela serão atualizadas. Quando está presente, apenas as linhas atendendo a condição escrita após o *WHERE* serão atualizadas. Observe que o sinal de igual na cláusula *SET* é uma atribuição, enquanto o sinal de igual na cláusula *WHERE* é uma comparação, mas isto não causa ambigüidade.

Também pode ser atualizada mais de uma coluna pelo comando *UPDATE*, colocando mais de uma atribuição na cláusula *SET*. Por exemplo:

```
UPDATE tsuitipoorgaocolegiado
SET strtipoorgaocol = ‘ORGÃOS COLEGIADOS DA ADMINISTRAÇÃO
CENTRAL’, chrtipo = ‘L’
WHERE numtipoorgaocol = 1;
```

10.1. Transações

Transação é um conceito fundamental de todo o sistema de banco de dados. O ponto essencial de uma transação é que esta engloba vários passos em uma única operação de tudo ou nada. Os estados dos passos intermediários não são visíveis para as outras transações concorrentes e, se alguma falha ocorrer que impeça a transação de chegar até o fim, então nenhum dos passos intermediários irá afetar o banco de dados de nenhuma forma.

No PostgreSQL uma transação é definida cercando-se os comandos SQL da transação com os comandos BEGIN e COMMIT. Sendo assim, a nossa transação bancária ficaria:

```
BEGIN;

     UPDATE conta_corrente
     SET saldo = saldo - 100.00
     WHERE nome = ’Alice’;

     -- etc etc

COMMIT;
```

Se no meio da transação for decidido que esta não deve ser concluída (talvez porque foi visto que o saldo da Alice se tornou negativo), pode ser executado o comando ROLLBACK em vez do COMMIT, fazendo com que todas as atualizações sejam canceladas.

O PostgreSQL na verdade trata todo comando SQL como sendo executado dentro de uma transação. Se não for utilizado o comando BEGIN, então cada comando possui individualmente um BEGIN e um (se der tudo certo) COMMIT em torno dele. Um grupo de comandos envolvidos por um BEGIN e um COMMIT é algumas vezes chamado de *bloco de transação*.

Nota: Algumas bibliotecas cliente emitem um comando BEGIN e um comando COMMIT automaticamente, fazendo com que se obtenha o efeito de um bloco de transação sem que seja perguntado. Verifique a documentação da interface sendo utilizada.

11. Excluindo linhas da tabela

Até aqui se mostrou como adicionar dados em tabelas, e como modificar estes dados. Está faltando mostrar como remover os dados que não são mais necessários. Assim como só é possível adicionar dados para toda uma linha, também uma linha só pode ser removida por completo de uma tabela. Na seção anterior foi visto que o SQL não oferece funcionalidade para endereçar diretamente linhas específicas. Portanto, a remoção de linhas só pode ser feita por meio da especificação das condições que as linhas a serem removidas devem atender. Havendo uma chave primária na tabela, então é possível especificar exatamente a linha. Mas também pode ser removido um grupo de linhas atendendo a uma determinada condição, ou podem ser removidas todas as linhas da tabela de uma só vez.

É utilizado o comando *DELETE* para excluir linhas; a sintaxe deste comando é muito semelhante a do comando *UPDATE*. Por exemplo, para excluir todas as linhas da tabela de Tipos de Órgãos Colegiados igual a 10, usa-se:

```
DELETE FROM tsuitipoorgaocolegiado
WHERE numtipoorgaocol = 10;
```

Se for escrito simplesmente.

```
DELETE FROM tsuitipoorgaocolegiado;
```

então todas as linhas da tabela serão excluídas!

Parte B: Avançado

12. Expressão condicional

12.1. CASE

```
CASE WHEN condição THEN resultado

     [WHEN ...]

     [ELSE resultado]

END
```

A expressão CASE do SQL é uma expressão condicional genérica, semelhante às declarações *if/else* de outras linguagens. A cláusula CASE pode ser empregada sempre que a utilização de uma expressão for válida. A *condição* é uma expressão que retorna um resultado booleano. Se a *condição* for verdade, então o valor da expressão CASE é o *resultado*. Se a *condição* for falsa, todas as cláusulas WHEN seguintes são percorridas da mesma maneira. Se nenhuma *condição* WHEN for verdade, então o valor da expressão CASE é o valor do *resultado* na cláusula ELSE. Se a cláusula ELSE for omitida, e nenhuma condição for satisfeita, o resultado será nulo.

Um exemplo:

```
=> SELECT chrtipo FROM sui.tsuitipoorgaocolegiado;

 chrtipo
----------
 C
 L
 C
 L
 L
 L
 C
 L
 L
 L
 L
 C
 C
```

```
=> SELECT chrtipo,
          CASE WHEN chrtipo = 'C' THEN 'Central'
               WHEN chrtipo = 'L' THEN 'Local'
               ELSE 'outro'
          END AS Tipo
    FROM sui.tsuitipoorgaocolegiado;
```

```
 chrtipo       | Tipo
----------------+-------
C              | Central
L              | Local
C              | Central
L              | Local
L              | Local
L              | Local
C              | Central
L              | Local
L              | Local
L              | Local
L              | Local
C              | Central
C              | Central
```

Os tipos de dado de todas as expressões para *resultado* devem poder ser convertidos no mesmo tipo de dado de saída.

```
     CASE expressão
         WHEN valor THEN resultado
         [WHEN ...]
         [ELSE resultado]
     END
```

Esta expressão CASE "simplificada" é uma variante especializada da forma geral mostrada acima. A *expressão* é calculada e comparada com todos os *valor*es das cláusulas WHEN, até ser encontrado um igual. Se nenhum valor igual for encontrado, o *resultado* na cláusula ELSE (ou o valor nulo) é retornado. Esta forma é semelhante à declaração switch da linguagem C.

O exemplo mostrado anteriormente pode ser escrito utilizando a sintaxe simplificada do CASE:

```
=> SELECT chrtipo,
          CASE chrtipo   WHEN 'C' THEN 'Central'
                         WHEN 'L' THEN 'Local'
                         ELSE 'outro'
          END AS Tipo
  FROM sui.tsuitipoorgaocolegiado;
```

12.2. COALESCE

```
COALESCE (expressao1, expressao2,  ...  expressaon)
```

A função COALESCE retorna o primeiro de seus argumentos que **não for nulo**. Geralmente é útil para substituir o valor padrão dos valores nulos quando os dados são usados para exibição. Por exemplo:

```
SELECT COALESCE (loc_numlocalidade, strceplocalidade,
'não definido') AS Localidade
FROM sui.tsuilocalidade
```

12.3. NULLIF

NULLIF(valor1, valor2)

A função NULLIF retorna o valor nulo se, e somente se, *valor1* e *valor2* forem iguais. Senão, retorna *valor1*. Pode ser utilizado para realizar a operação inversa do exemplo para COALESCE mostrado acima:

```
SELECT NULLIF(loc_numlocalidade, 'não definido')
FROM sui.tsuilocalidade
```

Dica: Tanto o COALESCE como o NULLIF são apenas formas abreviadas das expressões CASE. Na realidade, são convertidos em expressões CASE em um estágio bem inicial do processamento, fazendo o processamento subseqüente supor que está lidando com o CASE. Por isso, a utilização incorreta do COALESCE ou do NULLIF pode produzir uma mensagem de erro fazendo referência ao CASE.

13. Combinação de consultas

Os resultados de duas consultas podem ser combinados utilizando as operações com conjuntos [1] união (*union*), interseção (*intersection*) e diferença (*difference*). A sintaxe é

```
consulta1 UNION [ALL] consulta2

consulta1 INTERSECT [ALL] consulta2

consulta1 EXCEPT [ALL] consulta2
```

onde *consulta1* e *consulta2* são consultas que podem utilizar qualquer uma das funcionalidades discutidas anteriormente. As operações com conjuntos também podem ser aninhadas ou encadeadas.

```
consulta1 UNION consulta2 UNION consulta3

significa, na verdade,

(consulta1 UNION consulta2) UNION consulta3
```

UNION anexa o resultado da *consulta2* ao resultado da *consulta1* (embora não seja garantido que as linhas retornem nesta ordem). Além disso, são eliminadas todas as linhas duplicadas, do mesmo modo que no *DISTINCT*, a não ser que *UNION ALL* seja utilizado.

INTERSECT retorna todas as linhas presentes tanto no resultado da *consulta1* quanto no resultado da *consulta2*. As linhas duplicadas são eliminadas, a não ser que *INTERSECT ALL* seja utilizado.

EXCEPT retorna todas as linhas presentes no resultado da *consulta1*, mas que não estão presentes no resultado da *consulta2* (às vezes isto é chamado de *diferença* entre os dois resultados). Novamente, as linhas duplicadas são eliminadas a não ser que *EXCEPT ALL* seja utilizado.

Para ser possível calcular a união, a interseção, ou a diferença entre duas consultas, estas duas consultas precisam ser "compatíveis para união", significando que as duas devem retornar o mesmo número de colunas, e que as colunas correspondentes devem possuir tipos de dado compatíveis.

Notas

[1] Dados dois conjuntos A e B: chama-se *diferença* ente A e B o conjunto formado pelos elementos de A que não pertencem a B; chama-se *interseção* de A com B o conjunto formado pelos elementos comuns ao conjunto A e ao conjunto B; chama-se *união* de A com B o conjunto

formado pelos elementos que pertencem a A ou B. Edwaldo Bianchini e Herval Paccola - Matemática - Operações com conjuntos. (N.T.)

Exemplo:

```
SELECT strnomepaisuf AS Localidade
FROM sui.tsuiuf
WHERE strnomepaisuf LIKE 'BRASIL/S%'
UNION
SELECT strnomelocalidade AS Localidade
FROM sui.tsuilocalidade
WHERE strnomelocalidade LIKE 'S%'
```

```
Localidade
------------------------------------------------------------
BRASIL/SANTA CATARINA
BRASIL/SERGIPE
BRASIL/SÃO PAULO
SABARA
SABAUDIA
SABAUNA
SABIAGUABA
SABINO
SABINOPOLIS
SABOEIRO
...
```

14. Junção de tabelas

Até agora nossas consultas somente acessaram uma tabela de cada vez. As consultas podem acessar várias tabelas de uma vez, ou acessar a mesma tabela de uma maneira que várias linhas da tabela são processadas ao mesmo tempo. Uma consulta que acessa várias linhas da mesma tabela ou de tabelas diferentes de uma vez é chamada de consulta de *junção*.

14.1. Junção INNER JOIN (juncão interna)

Relacionam-se as colunas de chave estrangeira e chave primária da tabela.

```
SELECT *
FROM sui.tsuientidade, sui.tsuiua
WHERE numentidade = ent_numentidade
AND strnomeent LIKE 'FACULDADE%';
```

Uma vez que todas as colunas possuem nomes diferentes, o analisador encontra automaticamente a tabela que a coluna pertence, mas é um bom estilo qualificar completamente os nomes das colunas nas consultas de junção:

```
SELECT ent.strnomeent AS Unidade,
       ua.strnomeua AS Departamento
FROM sui.tsuientidade ent, sui.tsuiua ua
WHERE ent.numentidade = ua.ent_numentidade
AND ent.strnomeent LIKE 'FACULDADE%';
```

As consultas de junção do tipo visto até agora também poderiam ser escritas da seguinte forma alternativa:

```
SELECT *
FROM sui.tsuientidade INNER JOIN sui.tsuiua ON
(tsuientidade.numentidade = tsuiua.ent_numentidade);
```

A utilização desta sintaxe não é tão comum quanto das demais acima, mas é mostrada aqui para que sejam entendidos os próximos tópicos.

14.2. Junção OUTER JOIN (junção externa)

Se nenhuma linha for encontrada, nós queremos que algum “valor vazio” seja colocado nas colunas da tabela TSUIPROGRAMA. Este tipo de consulta é chamado de *junção externa* (outer join).

A combinação de junção externa é utilizada quando uma das linhas não é idêntica a da outra tabela relacionada.

O comando então fica assim:

```
SELECT o.strnomeorgaocol, o.prg_numprogramap,
       o.prg_numentidade
FROM sui.tsuiprograma p
LEFT OUTER JOIN sui.tsuiorgaocolegiado o ON
(     p.numprogramap = o.prg_numprogramap
  AND p.ent_numentidade = o.prg_numentidade
)
WHERE o.strsiglaorgaocol = 'COPGB'
```

Esta consulta é chamada de *junção externa esquerda* (left outer join) porque a tabela mencionada à esquerda do operador de junção terá cada uma de suas linhas aparecendo na saída ao menos uma vez, enquanto que a tabela à direita vai ter somente as linhas que correspondem a alguma linha da tabela à esquerda aparecendo. Ao listar uma linha da tabela à esquerda, para a qual não existe nenhuma linha correspondente na tabela à direita, valores vazios (null) são colocados nas colunas da tabela à direita. Existe também a junção externa direita (right outer join)

14.3. Junção FULL OUTER JOIN (junção externa completa)

Traz todas as linhas da tabela à direita mesmo que não tenha na tabela do lado esquerdo, e também traz todas as linhas da tabela à esquerda mesmo que não tenha na tabela do lado direito.

Exemplo:

```
SELECT p.strprogramacapes, o.strnomeorgaocol,
       o.prg_numprogramap, o.prg_numentidade
FROM sui.tsuiorgaocolegiado o
     FULL OUTER JOIN sui.tsuiprograma p
ON
(     p.numprogramap = o.prg_numprogramap
  AND p.ent_numentidade = o.prg_numentidade
)
```

14.4. Junção SELF JOIN (Autojunção)

Também é possível fazer a junção da tabela com si mesma. Isto é chamado de *autojunção* (self join). Como exemplo, suponha que nós desejamos descobrir a Entidade superior de uma entidade. Nós precisamos comparar a coluna ent_numentidade de cada linha da Entidade com a coluna numentidade de todas as outras linhas da tabela Entidade.

Exemplo:

```
SELECT e.numentidade, e.strnomeent, e.ent_numentidade,
f.numentidade, f.strnomeent
FROM sui.tsuientidade e, sui.tsuientidade f
WHERE e.ent_numentidade = f.numentidade
```

15. Agrupando os dados

Após passar pelo filtro *WHERE*, a tabela de entrada derivada pode estar sujeita a agrupamento, utilizando a cláusula *GROUP BY*, e a eliminação de grupos de linhas, utilizando a cláusula *HAVING*.

```
SELECT lista_seleção
FROM ...
[WHERE ...]
GROUP BY referência_coluna_agrupamento [,
referência_coluna_agrupamento]...
```

15.1. GROUP BY

O GROUP BY condensa em uma única linha todas as linhas selecionadas que compartilham os mesmos valores para as colunas agrupadas. As funções de agregação, caso existam, são computadas através de todas as linhas que pertencem a cada grupo, produzindo um valor separado para cada grupo (enquanto que sem GROUP BY, uma função de agregação produz um único valor computado através de todas as linhas selecionadas). Quando GROUP BY está presente, não é válido uma expressão de saída do SELECT fazer referência a uma coluna não agrupada, exceto dentro de uma função de agregação, porque pode haver mais de um valor possível retornado para uma coluna não agrupada.

Um item do GROUP BY pode ser o nome de uma coluna da entrada, o nome ou o número ordinal de uma coluna da saída (expressão SELECT), ou pode ser uma expressão arbitrária formada pelos valores das colunas da entrada.

No caso de haver ambigüidade, o nome no GROUP BY vai ser interpretado como o sendo o nome de uma coluna da entrada, e não como o nome de uma coluna da saída.

Por exemplo:

```
=> SELECT uf.strnomepaisuf, uf.flgtipopaisuf
   FROM sui.tsuiuf uf;
```

```
 strnomepaisuf                          | flgtipopaisuf
----------------------------------------+--------------------
BRASIL/ALAGOAS                                  B
BRASIL/ACRE                                     B
BRASIL/AMAPÁ                                    B
BRASIL/AMAZONAS                                 B
AFEGANISTAO                                     E
AFRICA DO SUL                                   E
ALBANIA                                         E
ALEMANHA                                        E
ALTO VOLTA                                      E
ANDORRA                                         E
...
(263 rows)
```

```
=> SELECT flgtipopaisuf
   FROM sui.tsuiuf uf
   GROUP BY flgtipopaisuf;
```

```
 flgtipopaisuf
-------------------
        B
        E
(2 rows)
```

Na segunda consulta não poderia ser escrito *SELECT * FROM* sui.tsuiuf uf *GROUP BY* flgtipopaisuf, porque não existe um único valor da coluna *y* que poderia ser associado com cada grupo. As colunas agrupadas podem ser referenciadas na lista de seleção, porque possuem um valor constante conhecido para cada grupo.

De modo geral, se uma tabela é agrupada as colunas que não são usadas nos agrupamentos não podem ser referenciadas, exceto nas expressões de agregação. Um exemplo de expressão de agregação é:

```
=> SELECT flgtipopaisuf, COUNT(strnomepaisuf)
   FROM sui.tsuiuf uf
   GROUP BY flgtipopaisuf;

flgtipopaisuf | count
-----------------+---------
B               28
E               235

(2 rows)
```

Aqui *count()* é a função de agregação que calcula a quantidade de registros para o grupo todo.

Dica: Um agrupamento sem expressão de agregação na verdade computa o conjunto de linhas distintas de uma coluna. Também poderia ser obtido por meio da cláusula *DISTINCT*.

15.2. Funções de Agregação

Utilizada na cláusula SELECT ou na cláusula HAVING.

As funções são:

- COUNT: agregação para contar
- SUM: somar
- AVG: calcular a média
- MAX: valor máximo
- MIN: valor mínimo

Exemplo:

> Qual a menor data de ativação de órgão colegiado para cada Entidade?

```
SELECT MIN(datativorgaocol),  ent_numentidade
FROM sui.tsuiorgaocolegiado
GROUP BY ent_numentidade
```

15.3. A cláusula HAVING

A condição opcional HAVING possui a forma geral:

HAVING *expressão_booleana* onde *expresão_booleana* é a mesma que foi especificada para a cláusula WHERE.

HAVING especifica uma tabela agrupada derivada pela eliminação das linhas agrupadas que não satisfazem a *expressão_booleana*. HAVING é diferente de WHERE: WHERE filtra individualmente as linhas antes da aplicação do GROUP BY, enquanto HAVING filtra os grupos de linhas criados pelo GROUP BY.

Cada coluna referenciada na *expressão_booleana* deve referenciar, sem ambigüidade, uma coluna de agrupamento, a menos que a referência apareça dentro de uma função de agregação.

Se uma tabela for agrupada utilizando a cláusula *GROUP BY*, mas há interesse em alguns grupos apenas, a cláusula *HAVING* pode ser utilizada, da mesma

forma que a cláusula *WHERE*, para remover grupos da tabela agrupada. A sintaxe é:

```
SELECT lista_seleção
FROM ... [WHERE ...]
GROUP BY ... HAVING expressão_booleana
```

As expressões na cláusula *HAVING* podem fazer referência tanto a expressões agrupadas quanto a expressões não agrupadas (as quais necessariamente envolvem uma função de agregação).

Exemplo:

```
=> SELECT flgtipopaisuf, COUNT(strnomepaisuf)
     FROM sui.tsuiuf uf
     GROUP BY flgtipopaisuf
     HAVING flgtipopaisuf != 'B';
```

```
flgtipopaisuf | count
----------------+---------

 E           235

 (1 row)
```

```
=> SELECT flgtipopaisuf, COUNT(strnomepaisuf)
     FROM sui.tsuiuf uf
     GROUP BY flgtipopaisuf
     HAVING flgtipopaisuf < 'E';
```

```
flgtipopaisuf | count
----------------+---------
 B           28

 (1 row)
```

A cláusula *WHERE* seleciona linhas por uma coluna que não é agrupada, enquanto a cláusula *HAVING* restringe a saída para os grupos.

16. Expressões de subconsulta

Esta seção descreve as expressões de subconsulta em conformidade com o padrão SQL disponíveis no PostgreSQL. Todas as formas das expressões documentadas nesta seção retornam resultados booleanos (verdade/falso).

A subconsulta é a combinação de uma consulta dentro de outra consulta.

O resultado da **consulta interna** retorna um valor que é usado pela **consulta externa**.

A subconsulta pode ser usada nas cláusulas:

- WHERE
- HAVING
- FROM

16.1. EXISTS

```
EXISTS ( subconsulta )
```

O argumento do `EXISTS` é uma declaração `SELECT` arbitrária, ou uma *subconsulta*. A subconsulta é avaliada para determinar se retorna alguma linha. Se retornar pelo menos uma linha, o resultado de `EXISTS` é "verdade"; se a subconsulta não retornar nenhuma linha, o resultado de `EXISTS` é "falso".

A subconsulta pode fazer referência às variáveis da consulta que a envolve, que atuam como constantes durante a avaliação da subconsulta.

A subconsulta geralmente só é executada até ser determinado se pelo menos uma linha é retornada, e não até o fim. Não é sensato escrever uma subconsulta que tenha efeitos colaterais (tal como chamar uma função de seqüência); se o efeito colateral ocorrerá ou não pode ser difícil de saber.

Uma vez que o resultado depende apenas do fato de alguma linha ser retornada, e não do conteúdo desta linha, normalmente não há interesse no conteúdo da saída da subconsulta. Uma convenção usual de codificação, é escrever todos os testes de *EXISTS* na forma *EXISTS(SELECT 1 WHERE ...)*. Entretanto, existem exceções para esta regra, como as subconsultas que utilizam `INTERSECT`.

```
SELECT * FROM sui.tsuiprograma
WHERE EXISTS (SELECT 1 FROM sui.tsuicursog);
```

16.2. IN

expressão IN (subconsulta)

O lado direito desta forma do IN é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é avaliada e comparada com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se uma linha igual for encontrada no resultado da subconsulta. O resultado é "falso" se nenhuma linha igual for encontrada (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Quando a expressão à esquerda for nula, ou não havendo nenhum valor igual à direita, e pelo menos uma das linhas à direita for nula, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para as combinações booleanas de valores nulos.

Da mesma forma que em EXISTS, não é sensato supor que a subconsulta será executada até o fim.

(expressão [, expressão ...]) IN (subconsulta)

O lado direito desta forma do IN é uma subconsulta entre parênteses, que deve retornar tantas colunas quantas forem as expressões existentes na lista do lado esquerdo. As expressões do lado esquerdo são avaliadas e comparadas com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada alguma linha igual na subconsulta. O resultado é "falso" se nenhuma linha igual for encontrada (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Como usual, os valores nulos nas expressões ou nas linhas da subconsulta são combinados conforme as regras normais do SQL para expressões booleanas. Duas linhas são consideradas iguais se todos os membros correspondentes forem iguais e não nulos; as linhas não são iguais se algum membro correspondente for diferente e não nulo; caso contrário, o resultado da comparação da linha é desconhecido (nulo). Se os resultados de todas as linhas forem diferentes ou nulos, com pelo menos um nulo, então o resultado do IN é nulo.

Exemplo:

```
SELECT * FROM sui.tsuitipoorgaocolegiado t
WHERE t.numtipoorgaocol IN
     (SELECT tipoorgaocoleg_numtipoorgaocol
      FROM sui.tsuiorgaocolegiado c, sui.tsuiprograma p
      WHERE c.prg_numprogramap = p.numprogramap
      AND c.prg_numentidade = p.ent_numentidade);
```

16.3. NOT IN

expressão NOT IN (*subconsulta*)

O lado direito desta forma do NOT IN é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão do lado esquerdo é avaliada e comparada com cada linha do resultado da subconsulta. O resultado do NOT IN é "verdade" se somente linhas diferentes forem encontradas no resultado da subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se qualquer linha igual for encontrada.

Quando a expressão à esquerda for nula, ou não havendo nenhum valor igual à direita, e pelo menos uma linha da direita for nula, o resultado da construção NOT IN será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para as combinações booleanas de valores nulos.

Do mesmo modo que no EXISTS, não é sensato supor que a subconsulta será executada até o fim.

(*expressão* [, *expressão* ...]) NOT IN (*subconsulta*)

O lado direito desta forma do NOT IN é uma subconsulta entre parênteses, que deve retornar tantas colunas quantas forem as expressões existentes na lista do lado esquerdo. As expressões do lado esquerdo são avaliadas e comparadas com cada linha do resultado da subconsulta. O resultado do NOT IN é "verdade" se somente linhas diferentes forem encontradas na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha. O resultado é “falso” se uma linha igual for encontrada).

Como usual, os valores nulos nas expressões ou nas linhas da subconsulta são combinados conforme as regras normais do SQL para expressões booleanas. Duas linhas são consideradas iguais se todos os membros correspondentes forem iguais e não nulos; as linhas não são iguais se algum membro correspondente for diferente e não nulo; caso contrário, o resultado da comparação da linha é desconhecido (nulo). Se todos os resultados das linhas forem diferentes ou nulos, com pelo menos um nulo, então o resultado de NOT IN é nulo.

Exemplo:

```
SELECT * FROM sui.tsuitipoorgaocolegiado t
WHERE t.numtipoorgaocol NOT IN
     (SELECT tipoorgaocoleg_numtipoorgaocol
      FROM sui.tsuiorgaocolegiado c, sui.tsuiprograma p
      WHERE c.prg_numprogramap = p.numprogramap
      AND c.prg_numentidade = p.ent_numentidade)
```

17. View (Visões)

O comando `CREATE VIEW` cria uma visão. A visão não é fisicamente materializada. Em vez disso, uma regra é automaticamente criada (uma regra ON SELECT) para realizar as operações de SELECT na visão.

O comando `CREATE OR REPLACE VIEW` é semelhante, mas se uma visão com o mesmo nome existir então é substituída. Somente pode ser substituída uma visão por outra que produza um conjunto idêntico de colunas (ou seja, colunas com os mesmos nomes e os mesmos tipos de dado).

Se o nome do esquema for fornecido (por exemplo, `CREATE VIEW meu_esquema.minha_visao ...`) então a visão será criada no esquema especificado, senão será criada no esquema. O nome da visão deve ser diferente do nome de qualquer outra visão, tabela, seqüência ou índice no mesmo esquema.

Exemplo:

Uma visão consistindo do nome de todas as entidades do tipo ‘U’ (Unidade Universitária) com seus respectivos departamentos:

```
CREATE OR REPLACE VIEW
sui.vw_uni_universitaria_departamentos AS
SELECT e.numentidade, e.strnomeent, u.numua, u.strnomeua
FROM sui.tsuientidade e, sui.tsuiua u
WHERE e.numentidade = u.ent_numentidade
AND e.chrtpent = 'U'
```

Consultando a visão:

```
SELECT * FROM sui.vw_uni_universitaria_departamentos
WHERE numentidade = 76
```

```
numentidade | strnomeent                                  | numua  | strnomeua
-----------------------------------------------------------------------------------------------------------
76          INSTITUTO DE ARTES DE SAO PAULO   52000  CAMPUS DE SAO PAULO-IA
76          INSTITUTO DE ARTES DE SAO PAULO  52050  DEP DE ARTES PLASTICAS
76          INSTITUTO DE ARTES DE SAO PAULO  52051  DEP ARTES CENICAS ED
FUND COM
76           INSTITUTO DE ARTES DE SAO PAULO 52052 DEP DE MUSICA
76           INSTITUTO DE ARTES DE SAO PAULO 52100 DIVISAO TECNICA
ACADEMICA
.... (36 rows)
```

Use o comando DROP VIEW para excluir uma visão.

Exercícios

1) Criar uma tabela no Schema **SUI** com os dados abaixo:

   Nome lógico: Função / Tipo no colegiado

   Descrição: Permite armazenar as funções possíveis que um MCA (Membro da Comunidade Acadêmica) pode assumir num colegiado. Exemplos: Presidente, Representante da FAPESP – Titular, Representante Docente – Suplente, etc.

   Nome da Tabela: TSUIFUNCAO_TPORGAOCOL

Nome do Atributo	Significado	Tipo	Tamanho	Obrigatório
numFuncaoCol	Código do tipo de função no colegiado, dentro do SUI.	integer		X
toc_numtipoOrgaoCol	Código do tipo de órgão colegiado	integer		X
funcoleg_numFuncaoCol	Código da função no colegiado	integer		X
chrunico		char	1	X

2) Alterar a estrutura da tabela TSUIFUNCAO_TPORGAOCOL

   a) Criar uma nova coluna.

Nome do Atributo	Significado	Tipo	Req
numquantidade	Quantidade	integer	X

   b) Apagar a coluna **chrunico**

3) Adicionar algumas restrições na tabela TSUIFUNCAO_TPORGAOCOL

   a) Chave primária para a coluna **numFuncaoCol**

   b) Chave estrangeira para a coluna **toc_numtipoOrgaoCol** relacionando com a tabela TSUITIPOORGAOCOLEGIADO com a coluna **numtipoorgaocol**

   c) Chave estrangeira para a coluna **funcoleg_numFuncaoCol** relacionando com a tabela TSUIFUNCAOCOLEGIADO com a coluna **numfuncaocol**

4) Inserir dados na tabela TSUIFUNCAO_TPORGAOCOL

numFuncaoCol	1
toc_numtipoOrgaoCol	Código do Coordenador
funcoleg_numFuncaoCol	Código de CONSELHOS DE PROGRAMAS DE POS GRADUAÇÃO

numFuncaoCol	2
toc_numtipoOrgaoCol	Código do Coordenador
Funcoleg_numFuncaoCol	Código de REPRESENTANTE DISCENTE DA POS GRADUAÇAO

5) Consultar os dados da tabela TSUIUA e realizar as seguintes tarefas:

    a) Mostrar na cláusula SELECT as colunas numua, ent_numentidade e strnomeua da tabela e nomear cada coluna da tabela com um álias.
    b) Com a consulta acima filtrar os dados trazendo somente as Unidades Administrativas cujo tipo (chrtpua) seja ‘D’ (Departamento) e que tenham o campo (strhistoriaua) preenchido.

6) Consultar dados da tabela TSUIMCA realizando a seguinte tarefa:

    a) Trazer somente as colunas nummca, strnomemca, chrtipomca, strconjugemca e strpaimca.
    b) Filtrar os dados pelo campo nummca trazendo entre os valores 18 a 500.
    c) Filtrar os dados trazendo somente os que possuem estado civil (chrestcivilmca) diferente de 1 (solteiro).
    d) Filtrar os dados trazendo somente os registros que não tenha o nome do cônjuge preenchido (strconjugemca) ou o nome do pai não preenchido (strpaimca).
    e) Ordenar as linhas em ordem decrescente pelo campo (chrtipomca).

7) Consultar dados da tabela TSUIENTIDADE realizando as seguintes tarefas:

    a) Trazer o nome da Entidade (strnomeent)
    b) Trazer a quantidade de dias ativos da Entidade até a data de desativação. Utilizar os campos (datativacao) e (datdesativ).
    c) Ordenar pelo número de dias Ativos.

8) Consultar dados da tabela TSUITIPOORGAOCOLEGIADO utilizando as seguintes funções para o campo (strtipoorgaocol):

a) Retornar o número de caracteres do campo.
b) Extrair parte da string do campo, do 1º ao 5º caracter
c) Preencher a string “**********” à esquerda até completar o tamanho de 46 caracteres com o preenchimento do campo especificado acima.
d) Remover de ambas as extremidades do campo os caracteres “CO”

9) Consultar dados da tabela TSUIMCA realizando as seguintes tarefas:

a) Crie uma consulta que retorne todos MCA’s que possuam o nome (strnomemca) com as iniciais “ADA”

b) Crie uma nova consulta nesta mesma tabela que retorne todos os MCA’s que possuam em seu nome completo (strnomemca) a palavra “SILVA”

c) Crie uma nova consulta nesta mesma tabela que retorne todos os MCA’s que possuam a 3ª letra “A” em seu nome (strnomemca).

10) Consultar dados da tabela TSUIENTIDADE realizando as seguintes tarefas:

a) Trazer somente as Entidades com tipo (chrtpent) C (Campus Universitário).
b) Filtrar os dados trazendo somente os que tiveram data ativada (datativacao) entre as datas '01/01/2000' e '01/12/2002', utilizar a função de conversão de tipo no campo data.
c) Ordenar as linhas em ordem crescente pelo campo (strnomeent).

11) Consultar dados da tabela TSUIORGAOCOLEGIADO trazendo a data de ativação (datativorgaocol) em formato completo (dia, mês por extenso e ano).

12) Na tabela TSUIORGAOCOLEGIADO, alterar o valor do atributo (strstatus) para ‘U’ e a data de desativação (datdesativorgaocol) para ‘30/08/2005’, cujo atributo que representa o código da Entidade (ent_numentidade) seja igual a 70.

13) Na tabela TSUIORGAOCOLEGIADO, excluir os dados da tabela cujo atributo (strfunlegalorgaocol) esteja em branco.

14) Utilizar a query do exercício nº 6 trazendo somente os campos do nome da Entidade (strnomeent) e o tipo da Entidade (chrtpent). Neste último campo utilizar a expressão condicional CASE trazendo a descrição do tipo de cada entidade e não o código. Utilizar os seguintes valores:

- C = Campus Universitário

- T = Campus Complexo

- R = Unidades Complementares

15) Fazer uma combinação de consultas utilizando a operação de conjunto UNION da seguinte forma:

- No primeiro conjunto trazer as Unidades Universitárias. Utilizar a tabela TSUIENTIDADE cujo atributo tipo (chrtpent) seja igual à ‘U’. Trazer somente os atributos numentidade e strnomeent da tabela e atribuir um álias para cada coluna com o nome “codigo” e “descricao”, respectivamente.

- No segundo conjunto trazer as unidades administrativas cujo tipo seja Departamento. Utilizar a tabela TSUIUA cujo atributo tipo (chrtpua) seja igual à ‘D’. Trazer somente os atributos numua e strnomeua da tabela e atribuir um álias para cada coluna com o nome “codigo” e “descricao”, respectivamente.

16) Fazer uma junção simples (INNER JOIN) com as tabelas TSUIFUNCAOCOLEGIADO, TSUIFUNCAO_TPORGAOCOL e TSUITIPOORGAOCOLEGIADO. Trazer somente os campos numfuncaocol, strfuncaocol, numtipoorgaocol e strtipoorgaocol.

17) Fazer um agrupamento de dados utilizando a tabela TSUIENTIDADE realizando as seguintes tarefas (fazer um exercício por vez para ver a diferença):

a) Utilizar a função de agregação count().
b) Agrupar os dados pelo tipo da entidade (chrtpent).
c) Trazer somente os dados cujo campo sigla (strsiglaent) esteja preenchido.
d) Trazer somente os dados cuja quantidade de tipos seja menor que 10.

18) Fazer uma consulta retornando os dados da tabela TSUITIPOORGAOCOLEGIADO desde que não exista nenhum tipo de órgão colegiado cadastrado na TSUIORGAOCOLEGIADO

19) Fazer uma view que contenha os dados dos Cursos de Graduação em suas respectivas Entidades, fazer da seguinte forma:

a) Utilizar as tabelas TSUICURSOG, TSUICURSOGERAL e TSUIENTIDADE.
b) Trazer somente o código e nome da unidade, código e nome do curso de graduação.