UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
Escola Agrícola de Jundiaí

Banco de Dados

Apostila 03 – Utilizando SQL - DDL e DML, e Uso de Funções

Prof° Carlos Henrique Grilo Diniz

Macaíba, julho de 2019.

Índice

Índice de Figuras

1. Noções Básicas da Linguagem ANSI SQL

1.1. 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.

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

Nesta etapa iremos aprender a como criar um banco de dados com a linguagem 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 a linguagem DML (Data Manipulation Language) - Linguagem que define os comandos de manipulação e operação dos dados (comandos de consulta e atualização dos dados das tabelas).

1.1.1. O que é PostgreSQL ?

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional (SGBDOR), baseado no POSTGRES Versão 4.2 (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html) desenvolvido pelo Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley. O POSTGRES foi pioneiro em vários conceitos que tornaram disponíveis muito mais tarde em alguns sistemas de banco de dados comerciais.

O PostgreSQL é um descendente de código fonte aberto deste código original de Berkeley. É suportada grande parte do padrão SQL:2003, além de serem oferecidas muitas funcionalidades modernas, como:

- comandos complexos
- chaves estrangeiras
- gatilhos
- visões
- integridade transacional
- controle de simultaneidade multiversão

Além disso, o PostgreSQL pode ser estendido pelo usuário de muitas maneiras como, por exemplo, adicionando novos

- tipos de dado
- funções
- operadores
- funções de agregação

- métodos de índice
- linguagens procedurais

Devido à sua licença liberal, o PostgreSQL pode ser utilizado, modificado e distribuído por qualquer pessoa para qualquer finalidade, seja privada, comercial ou acadêmica, livre de encargos.

1.2. Tipos de dados

Nas tabelas que se seguem serão apresentados os principais tipos de dados que são utilizados no PostgreSQL. Entre estes tipos de dados temos os numéricos, os com suporte ao armazenamento de caracteres, data e hora, Lógico (Booleando).

Numéricos

Os tipos numéricos consistem em inteiros de dois, quatro e oito bytes, números de ponto flutuante de quatro e oito bytes, e decimais de precisão selecionável (tipos decimal e numeric).

Nome do Tipo Numérico	Tamanho (Bytes)	Sinônimo	Faixa de Valores
SMALLINT	02	INT2	-32768 a +32767
INTEGER	04	INT4	-2147483648 a +2147483647
BIGINT	08	INT8	-9223372036854775808 a 9223372036854775807
DECIMAL	VARIÁVEL	DECIMAL	sem limite
NUMERIC	VARIÁVEL	NUMERIC	sem limite
REAL	04	FLOAT4	precisão de 6 dígitos decimais
DOUBLE PRECISION	08	FLOAT8	precisão de 15 dígitos decimais
SERIAL	04	SERIAL	1 a 2147483647
BIGSERIAL	08	BIGSERIAL	1 a 9223372036854775807

Figura 1 – Tabela Contendo os principais tipos numéricos do PostgreSQL

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 ocasionará um erro.

O tipo integer, ou INT4, 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, ou INT8, 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, ou FLOAT4 ou FLOAT8 respectivamente, 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.

Caracteres

Nome do Tipo Caracter	Tamanho (Bytes)	Sinônimo
character varying(n)	VARIÁVEL	varchar(n)
character(n)	VARIÁVEL	char(n)
text	VARIÁVEL	Text

Figura 2 – Tabela Contendo os tipos caracteres do PostgreSQL

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. 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 (sinônimos) 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.

Data e Hora Simplificado

O PostgreSQL suporta o conjunto completo de tipos para data e hora do SQL, mostrados na Figura 3.

Tabela 8-9. Tipos para data e hora simplificados

Nome	Tamanho (Bytes)	Descrição	Menor valor	Maior valor	Resolução
interval [(p)]	12 bytes	intervalo de tempo	-178000000 anos	178000000 anos	1 microssegundo / 14 dígitos
date	4 bytes	somente data	4713 AC	32767 DC	1 dia
time [(p)] [without time zone]	8 bytes	somente a hora do dia	00:00:00.00	23:59:59.99	1 microssegundo / 14 dígitos
time [(p)] with time zone	12 bytes	somente a hora do dia, com zona horária	00:00:00.00+12	23:59:59.99-12	1 microssegundo / 14 dígitos

Figura 3 – Tabela Contendo os tipos de Data e Hora do PostgreSQL

2. Primeiros Passos na Linguagem ANSI SQL

2.1. Conceitos

O PostgreSQL é um sistema de gerenciamento de banco de dados relacional (SGBDR). Isto significa que é um sistema para gerenciar dados armazenados em relações. Relação é, essencialmente, um termo matemático para tabela. A noção de armazenar dados em tabelas é tão trivial hoje em dia que pode parecer totalmente óbvio, mas existem várias outras formas de organizar bancos de dados. Arquivos e diretórios em sistemas operacionais tipo Unix são um exemplo de banco de dados hierárquico. Um desenvolvimento mais moderno são os bancos de dados orientados a objeto.

Cada tabela é uma coleção nomeada de linhas. Todas as linhas de uma determinada tabela possuem o mesmo conjunto de colunas nomeadas, e cada coluna é de um tipo de dado específico. Enquanto as colunas possuem uma ordem fixa nas linhas, é importante lembrar que o SQL não garante a ordem das linhas dentro de uma tabela (embora as linhas possam ser explicitamente ordenadas para a exibição).

As tabelas são agrupadas em bancos de dados, e uma coleção de bancos de dados gerenciados por uma única instância do servidor PostgreSQL forma um agrupamento de bancos de dados.

2.2. Criação de tabelas (Comando de DDL)

Pode-se criar uma tabela especificando o seu nome juntamente com os nomes das colunas e seus tipos de dado:

```
CREATE TABLE clima (
    cidade          varchar(60),
    temp_min        int,            -- temperatura mínima
    temp_max        int,            -- temperatura máxima
    prcp            float4,         -- precipitação
    data            date
);
```

Espaços em branco (ou seja, espaços, tabulações e novas linhas) podem ser utilizados livremente nos comandos SQL. Isto significa que o comando pode ser digitado com um alinhamento diferente do mostrado acima, ou mesmo tudo em uma única linha. Dois hífens (“--”) iniciam um comentário; tudo que vem depois é ignorado até o final da linha. A linguagem SQL não diferencia letras maiúsculas e minúsculas nas palavras chave e nos identificadores, a não ser que os identificadores sejam colocados entre aspas (") para preservar letras maiúsculas e minúsculas, o que não foi feito acima.

No comando, varchar(60) especifica um tipo de dado que pode armazenar cadeias de caracteres arbitrárias com comprimento até 60 caracteres; int é o tipo inteiro normal; float4 é o tipo para armazenar números de ponto flutuante; date é o tipo para armazenar

data e hora (a coluna do tipo date pode se chamar date, o que tanto pode ser conveniente quanto pode causar confusão).

O PostgreSQL suporta os tipos SQL padrão int, smallint, float4, double precision, char(N), varchar(N), date, time, timestamp e interval, assim como outros tipos de utilidade geral, e um conjunto abrangente de tipos geométricos. O PostgreSQL pode ser personalizado com um número arbitrário de tipos definidos pelo usuário. Como conseqüência, sintaticamente os nomes dos tipos não são palavras chave, exceto onde for requerido para suportar casos especiais do padrão SQL.

No segundo exemplo são armazenadas cidades e suas localizações geográficas associadas:

```
CREATE TABLE cidades (
    nome            varchar(60),
    localizacao     point
);
```

O tipo point é um exemplo de tipo de dado específico do PostgreSQL.

Para terminar deve ser mencionado que, quando a tabela não é mais necessária, ou se deseja recriá-la de uma forma diferente, é possível removê-la por meio do comando:

```
DROP TABLE nome_da_tabela;
```

2.3. Inserção de linhas em tabelas (Comando de DML)

É utilizado o comando INSERT para inserir linhas nas tabelas:

```
INSERT INTO clima VALUES ('Natal', 20, 34, 10.5, '2013-08-09');
```

Repare que todos os tipos de dado possuem formato de entrada de dados bastante óbvios. As constantes, que não são apenas valores numéricos, geralmente devem estar entre apóstrofos ('), como no exemplo acima. O tipo date é, na verdade, muito flexível em relação aos dados que aceita, mas para este tutorial vamos nos fixar no formato sem ambigüidade mostrado acima.

O tipo point requer um par de coordenadas como entrada, como mostrado abaixo:

```
INSERT INTO cidades VALUES ('Natal', '(6.0, 40.0)');
```

A sintaxe usada até agora requer que seja lembrada a ordem das colunas. Uma sintaxe alternativa permite declarar as colunas explicitamente:

```
INSERT INTO clima (cidade, temp_min, temp_max, prcp, data)
    VALUES ('Natal', 21, 29, 17.8, '2013-07-08');
```

Se for desejado, pode-se declarar as colunas em uma ordem diferente, e pode-se, também, omitir algumas colunas. Por exemplo, se a precipitação não for conhecida:

```
INSERT INTO clima (data, cidade, temp_max, temp_min)
    VALUES ('2013-08-09', 'Macaiba', 19, 32);
```

Muitos desenvolvedores consideram declarar explicitamente as colunas um estilo melhor que confiar na ordem implícita.

Também pode ser utilizado o comando COPY para carregar uma grande quantidade de dados a partir de arquivos texto puro. Geralmente é mais rápido, porque o comando COPY é otimizado para esta finalidade, embora possua menos flexibilidade que o comando INSERT. Para servir de exemplo:

```
COPY clima FROM '/home/user/clima.txt';
```

O arquivo contendo os dados deve poder ser acessado pelo servidor e não pelo cliente, porque o servidor lê o arquivo diretamente. Podem ser obtidas mais informações sobre o comando COPY em COPY.

2.4. Consultar tabelas (Comando de DML)

Para trazer os dados de uma tabela, a tabela deve ser *consultada*. Para esta finalidade é utilizado o comando SELECT do SQL. Este comando é dividido em *lista de seleção* (a parte que especifica as colunas a serem trazidas), *lista de tabelas* (a parte que especifica as tabelas de onde os dados vão ser trazidos), e uma *qualificação opcional* (a parte onde são especificadas as restrições). Por exemplo, para trazer todas as linhas da tabela clima digite:

```
SELECT * FROM clima;
```

Resultado:

```
geografia=# SELECT * FROM clima;
 cidade  | temp_min | temp_max | prcp |    data
---------+----------+----------+------+------------
 Natal   |       20 |       34 | 10.5 | 2013-08-09
 Natal   |       21 |       29 | 17.8 | 2013-07-08
 Macaiba |       32 |       19 |      | 2013-08-09
(3 registros)
```

(aqui * é uma forma abreviada de "todas as colunas"). Pode-se obter o o mesmo resultados utilizando a seguinte sentença:

```
SELECT cidade, temp_min, temp_max, prcp, data FROM clima;
```

A saída deve ser:

```
geografia=# SELECT cidade, temp_min, temp_max, prcp, data FROM clima;
 cidade  | temp_min | temp_max | prcp |    data
---------+----------+----------+------+------------
 Natal   |       20 |       34 | 10.5 | 2013-08-09
 Natal   |       21 |       29 | 17.8 | 2013-07-08
 Macaiba |       32 |       19 |      | 2013-08-09
(3 registros)
```

Na lista de seleção podem ser especificadas expressões, e não apenas referências a colunas. Por exemplo, pode ser escrito)

```
SELECT cidade, (temp_max+temp_min)/2 AS temp_media, data FROM clima;
```

devendo produzir:

```
cidade  | temp_media |    data
---------+------------+------------
 Natal   |         27 | 2013-08-09
 Natal   |         25 | 2013-07-08
 Macaiba |         25 | 2013-08-09
(3 registros)
```

Perceba que a cláusula AS foi utilizada para mudar o nome da coluna de saída (a cláusula AS é opcional).

A consulta pode ser "qualificada", adicionando a cláusula WHERE para especificar as linhas desejadas. A cláusula WHERE contém expressões booleanas (valor verdade), e somente são retornadas as linhas para as quais o valor da expressão booleana for verdade. São permitidos os operadores booleanos usuais (AND, OR e NOT) na qualificação. Por exemplo, o comando abaixo retorna os registros do clima de São Francisco nos dias de chuva:

```
SELECT * FROM clima WHERE cidade = 'Natal' AND prcp > 0.0;
```

Resultado:

```
cidade | temp_min | temp_max | prcp |    data
--------+----------+----------+------+------------
 Natal  |       20 |       34 | 10.5 | 2013-08-09
 Natal  |       21 |       29 | 17.8 | 2013-07-08
(2 registros)
```

Pode ser solicitado que os resultados da consulta sejam retornados em uma determinada ordem:

```
SELECT * FROM clima
    ORDER BY cidade;
```

Resultado:

```
cidade  | temp_min | temp_max | prcp |    data
---------+----------+----------+------+------------
 Macaiba |       32 |       19 |      | 2013-08-09
 Natal   |       20 |       34 | 10.5 | 2013-08-09
 Natal   |       21 |       29 | 17.8 | 2013-07-08
(3 registros)
```

Pode ser solicitado que as linhas duplicadas sejam removidas do resultado da consulta:

```
SELECT DISTINCT cidade
    FROM clima;
```

Resultado:

```
cidade
---------
 Macaiba
 Natal
(2 registros)
```

Novamente, neste exemplo a ordem das linhas pode variar. Pode-se garantir resultados consistentes utilizando DISTINCT e ORDER BY juntos:

```
INSERT INTO clima VALUES ('Monte das Gameleiras', 16, 35, 0.0, '2013-07-08');
```

```
SELECT * FROM clima cidade;
```

Resultado:

```
        cidade        | temp_min | temp_max | prcp |    data
----------------------+----------+----------+------+------------
 Natal                |       20 |       34 | 10.5 | 2013-08-09
 Natal                |       21 |       29 | 17.8 | 2013-07-08
 Macaiba              |       32 |       19 |      | 2013-08-09
 Monte das Gameleiras |       16 |       35 |    0 | 2013-07-08
(4 registros)
```

```
SELECT DISTINCT cidade FROM clima;
```

Resultado:

```
geografia=# SELECT * FROM clima ORDER BY cidade;
        cidade        | temp_min | temp_max | prcp |    data
----------------------+----------+----------+------+------------
 Macaiba              |       32 |       19 |      | 2013-08-09
 Monte das Gameleiras |       16 |       35 |    0 | 2013-07-08
 Natal                |       20 |       34 | 10.5 | 2013-08-09
 Natal                |       21 |       29 | 17.8 | 2013-07-08
(4 registros)
```

2.5. Junções entre tabelas

Até agora as 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 sejam processadas ao mesmo tempo. A consulta que acessa várias linhas da mesma tabela, ou de tabelas diferentes, de uma vez, é chamada de consulta de *junção*. Como exemplo, suponha que se queira listar todas as linhas de clima junto com a localização da cidade associada. Para se fazer isto, é necessário comparar a coluna cidade de cada linha da tabela clima com a coluna nome de todas as linhas da tabela cidades, e selecionar os pares de linha onde estes valores são correspondentes.

> **Nota:** Este é apenas um modelo conceitual, a junção geralmente é realizada de uma maneira mais eficiente que comparar de verdade cada par de linhas possível, mas isto não é visível para o usuário.

Esta operação pode ser efetuada por meio da seguinte consulta:

```
SELECT * FROM clima, cidades WHERE cidade = nome;
```

Resultado:

```
cidade | temp_min | temp_max | prcp |    data    | nome  | localizacao
--------+----------+----------+------+------------+-------+-------------
 Natal  |       20 |       34 | 10.5 | 2013-08-09 | Natal | (6,40)
 Natal  |       21 |       29 | 17.8 | 2013-07-08 | Natal | (6,40)
(2 registros)
```

```
INSERT INTO cidades VALUES ('Monte das Gameleiras', '(6.1, 41.0)');
```

```
geografia=# SELECT * FROM clima, cidades WHERE cidade = nome;
   cidade        | temp_min | temp_max | prcp |    data    |        nome         | localizacao
---------------------+----------+----------+------+------------+---------------------+-----------
 Monte das Gameleiras |       16 |       35 |    0 | 2013-07-08 | Monte das Gameleiras | (6.1,41)
 Natal                |       20 |       34 | 10.5 | 2013-08-09 | Natal                | (6,40)
 Natal                |       21 |       29 | 17.8 | 2013-07-08 | Natal                | (6,40)
(3 registros)
```

```
SELECT cidade, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
    WHERE cidade = nome;
      cidade        | temp_min | temp_max | prcp |    data    | localizacao
---------------------+----------+----------+------+------------+-------------
 Monte das Gameleiras |       16 |       35 |    0 | 2013-07-08 | (6.1,41)
 Natal                |       20 |       34 | 10.5 | 2013-08-09 | (6,40)
 Natal                |       21 |       29 | 17.8 | 2013-07-08 | (6,40)
(3 registros)
```

Exercício em sala: Descobrir a semântica desta consulta quando a cláusula WHERE é omitida.

2.6. Funções de agregação (Executadas junto com o INSERT)

Como a maioria dos produtos de banco de dados relacional, o PostgreSQL suporta funções de agregação. Uma função de agregação computa um único resultado para várias linhas de entrada. Por exemplo, existem funções de agregação para contar (count), somar (sum), calcular a média (avg), o valor máximo (max) e o valor mínimo (min) para um conjunto de linhas.

Para servir de exemplo, é possível encontrar a maior temperatura mínima ocorrida em qualquer lugar usando

```
SELECT max(temp_min) FROM clima;
max
-----
  32
(1 registro)
```

Se for desejado saber a cidade (ou cidades) onde esta temperatura ocorreu pode-se tentar usar

```
SELECT cidade FROM clima WHERE temp_min = max(temp_min);      ERRADO
```

mas não vai funcionar, porque a função de agregação max não pode ser usada na cláusula WHERE (Esta restrição existe porque a cláusula WHERE determina as linhas que vão passar para o estágio de agregação e, portanto, precisa ser avaliada antes das funções de agregação serem computadas). Entretanto, como é geralmente o caso, a consulta pode ser reformulada para obter o resultado pretendido, o que será feito por meio de uma *subconsulta*:

```
SELECT cidade FROM clima
    WHERE temp_min = (SELECT max(temp_min) FROM clima);
cidade
---------
 Macaiba
(1 registro)
```

Isto está correto porque a subconsulta é uma ação independente, que calcula sua agregação em separado do que está acontecendo na consulta externa.

As agregações também são muito úteis em combinação com a cláusula GROUP BY. Por exemplo, pode ser obtida a maior temperatura mínima observada em cada cidade usando

```
SELECT cidade, max(temp_min)
    FROM clima
    GROUP BY cidade;
          cidade        | max
----------------------+-----
 Monte das Gameleiras |  16
 Natal                |  21
 Macaiba              |  32
(3 registros)
```

Produzindo uma linha de saída para cada cidade. Cada resultado da agregação é computado sobre as linhas da tabela correspondendo a uma cidade. As linhas agrupadas podem ser filtradas utilizando a cláusula HAVING

```
SELECT cidade, max(temp_min)
    FROM clima
    GROUP BY cidade
    HAVING max(temp_min) < 32;
```

```
        cidade       | max
---------------------+-----
 Monte das Gameleiras |  16
 Natal               |  21
(2 registros)
```

Que mostra os mesmos resultados, mas apenas para as cidades que possuem todos os valores de temp_min abaixo de 32. Para concluir, se desejarmos somente as cidades com nome começando pela letra "N" podemos escrever:

```
SELECT cidade, max(temp_min)
    FROM clima
    WHERE cidade LIKE 'N%'
    GROUP BY cidade
    HAVING max(temp_min) < 32;
```

```
 cidade | max
--------+-----
 Natal  |  21
(1 registro)
```

2.7. Atualizações (Comando de DML)

As linhas existentes podem ser atualizadas utilizando o comando UPDATE. Suponha que foi descoberto que as leituras de temperatura estão todas mais altas 2 graus. Os dados podem ser atualizados da seguinte maneira:

```
UPDATE clima
    SET temp_max = temp_max - 2,  temp_min = temp_min - 2
    WHERE data > '2013-08-09';
```

```
UPDATE 0
```

Agora vejamos o novo estado dos dados:

```
SELECT * FROM clima;
```

```
cidade        | temp_min | temp_max | prcp |    data
----------------------+----------+----------+------+------------
 Natal                |       20 |       34 | 10.5 | 2013-08-09
 Natal                |       21 |       29 | 17.8 | 2013-07-08
 Macaiba              |       32 |       19 |      | 2013-08-09
 Monte das Gameleiras |       16 |       35 |    0 | 2013-07-08
(4 registros)
```

2.8. Exclusões (Comando de DML)

As linhas podem ser removidas da tabela através do comando DELETE. Suponha que não estamos mais interessados nos registros do clima em **Monte das Gameleiras**. Então precisamos excluir estas linhas da tabela.

```
DELETE FROM clima WHERE cidade = 'Monte das Gameleiras';
```

```
DELETE 1
```

Todos os registros de clima pertencentes a Monte das Gameleiras são removidos.

```
SELECT * FROM clima;
```

```
cidade  | temp_min | temp_max | prcp |    data
---------+----------+----------+------+------------
 Natal   |       20 |       34 | 10.5 | 2013-08-09
 Natal   |       21 |       29 | 17.8 | 2013-07-08
 Macaiba |       32 |       19 |      | 2013-08-09
(3 registros)
```

Deve-se tomar cuidado com comandos na forma:

DELETE FROM *nome_da_tabela;*

! Sem uma qualificação, o comando DELETE remove *todas* as linhas da tabela, deixando-a vazia. O sistema não solicita confirmação antes de realizar esta operação!

2.9. Alteração de Tabelas (Comando de DDL)

Os campos podem ser acrescentados ou removidos de uma tabela com o comando ALTER TABLE. Suponha que seja necessário acrescentar o comando Unidade da federação a tabela clima. Então utilizamos o comando ALTER TABLE da seguinte forma:

```
ALTER TABLE clima ADD UF char(02);
```

O campo acima acrescenta o campo UF a tabela clima que é do tipo char com 02 caracteres.

2.10. Destruição de Tabelas (Comando de DDL)

As tabelas podem ser removidas dos Bancos de Dados utilizando-se o comando DROP TABLE. Para apagarmos uma tabela utilizamos o comando DROP TABLE **nome_da_tabela**.

2.11. Visões (Comando de DDL)

As visões são uma forma diferenciada de combinar, criar e consultar dados a partir de um conjunto de combinando os registros. No exemplo abaixo foram utilizados os dados de clima e de localização das cidades, atribuindo um nome a esta consulta pelo qual será possível referenciá-la como se fosse uma tabela comum.

```
CREATE VIEW minha_visao AS
    SELECT cidade, temp_min, temp_max, prcp, data, localizacao
        FROM clima, cidades
        WHERE cidade = nome;

SELECT * FROM minha_visao;
```

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL. As visões permitem encapsular, atrás de interfaces que não mudam, os detalhes da estrutura das tabelas, que podem mudar na medida em que as aplicações evoluem.

As visões podem ser utilizadas em praticamente todos os lugares onde uma tabela real pode ser utilizada. Construir visões baseadas em visões não é raro.

2.12. Chaves estrangeiras

Reveja as tabelas clima e cidades no Capítulo 2 . Considere o seguinte problema: Desejamos ter certeza que não serão inseridas linhas na tabela clima sem que haja um registro correspondente na tabela cidades. Isto é

chamado de manter a integridade referencial dos dados. Em sistemas de banco de dados muito simples poderia ser implementado (caso fosse) olhando primeiro a tabela cidades para verificar se existe a linha correspondente e, depois, inserir ou rejeitar a nova linha de clima. Esta abordagem possui vários problemas, e é muito inconveniente, por isso o PostgreSQL pode realizar esta operação por você.

A nova declaração das tabelas ficaria assim:

```
CREATE TABLE cidades (
    cidade      varchar(80) primary key,
    localizacao point
);

CREATE TABLE clima (
    cidade     varchar(80) references cidades(cidade),
    temp_min   int,
    temp_max   int,
    prcp       real,
    data       date
);
```

Operadores e Funções na Linguagem SQL

2.13. Operadores

Operador	Descrição
<	Menor que
>	Maior que
<=	Menor ou igual que
>=	Maior ou igual que
=	Igual
<> ou !=	Diferente

Operador	Descrição	Exemplo	Resultado
+	Adição	2 + 3	5
-	Subtração	2 - 3	-1
*	Multiplicação	2 * 3	6
/	Divisão inteira	4 / 2	2
%	Módulo	5 % 4	1
^	Exponenciação	2.0 ^ 3.0	8
!	Fatorial	5 !	120
@	Valor Absoluto	@ -5.0	5

2.14. Principais Funções Matemáticas

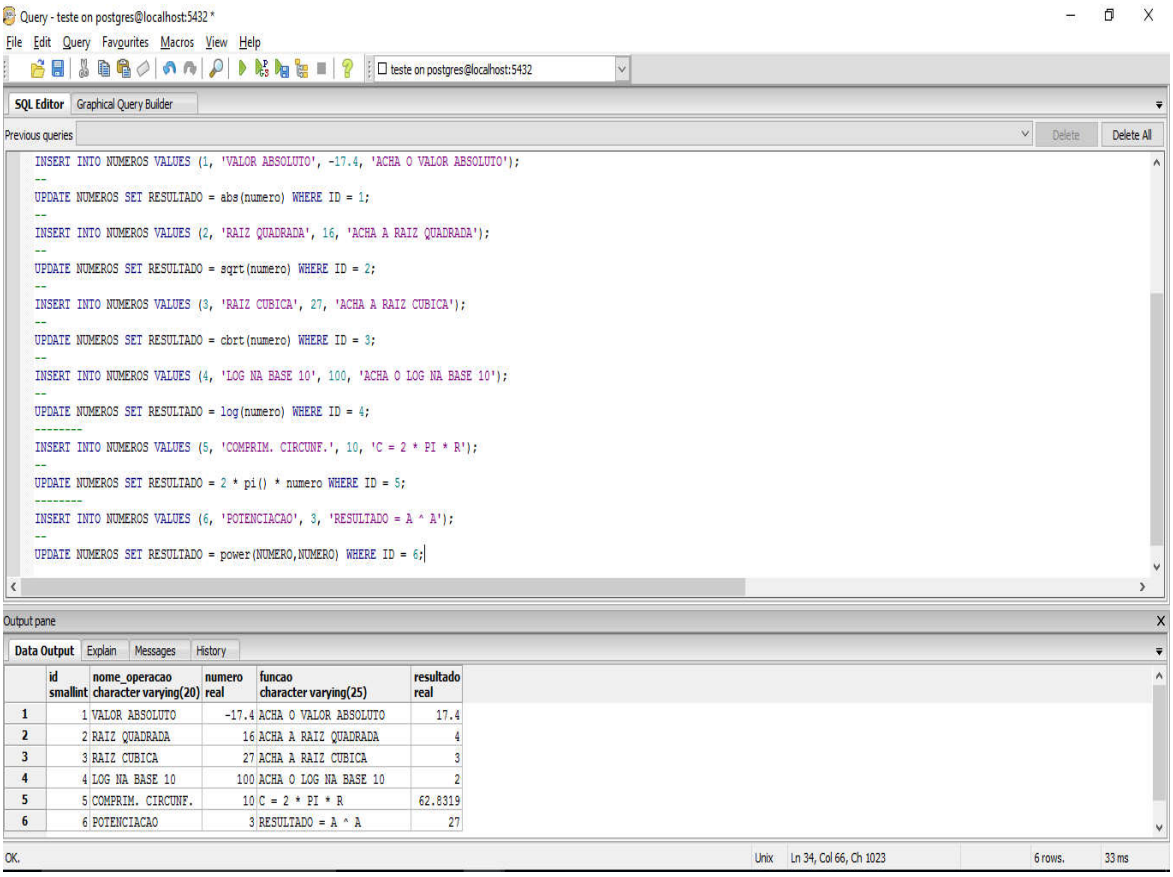
Função	Retorno e Descrição	Exemplo	Resultado
abs(x)	Valor absoluto	abs(-17.4)	17.4
sqrt(dp)	Raiz Quadrada	sqrt(16.0)	4.0
cbrt(dp)	Raiz Cúbica	cbrt(27.0)	3
log (dp)	Log na Base10	log(100)	2
Pi ()	Retorna o valor de "π"	pi ( )	3.14159265
power(a, b)	Retorna a^b	Power (3, 3)	27

```
CREATE TABLE NUMEROS (
        ID INT2,
        NOME_OPERACAO VARCHAR(20),
        NUMERO FLOAT4,
        FUNCAO VARCHAR(25),
        RESULTADO FLOAT4 );
```

```
SELECT * FROM NUMEROS;

INSERT INTO NUMEROS VALUES (1, 'VALOR ABSOLUTO', -17.4, 'ACHA O VALOR ABSOLUTO');
--
UPDATE NUMEROS SET RESULTADO = abs(numero) WHERE ID = 1;
--
INSERT INTO NUMEROS VALUES (2, 'RAIZ QUADRADA', 16, 'ACHA A RAIZ QUADRADA');
--
UPDATE NUMEROS SET RESULTADO = sqrt(numero) WHERE ID = 2;
--
INSERT INTO NUMEROS VALUES (3, 'RAIZ CUBICA', 27, 'ACHA A RAIZ CUBICA');
--
UPDATE NUMEROS SET RESULTADO = cbrt(numero) WHERE ID = 3;
--
```

```
INSERT INTO NUMEROS VALUES (4, 'LOG NA BASE 10', 100, 'ACHA O LOG NA BASE 10');
--
UPDATE NUMEROS SET RESULTADO = log(numero) WHERE ID = 4;
--------
INSERT INTO NUMEROS VALUES (5, 'COMPRIM. CIRCUNF.', 10, 'C = 2 * PI * R');
--
UPDATE NUMEROS SET RESULTADO = 2 * pi() * numero WHERE ID = 5;
--------
INSERT INTO NUMEROS VALUES (6, 'POTENCIACAO', 3, 'RESULTADO = A ^ A');
--
UPDATE NUMEROS SET RESULTADO = power(NUMERO,NUMERO) WHERE ID = 6;
--------
```



2.15. Principais Funções de manipulação de String

Função	Retorno e Descrição	Exemplo	Resultado
bit_length(string)	Int Tam.Bits	bit_length('jose')	32
char_length(string)	Int Tam.bytes	char_length('jose')	4
lower(string)	Converte p/ minusc.STR	lower('JOSE')	Jose
position(substring in string)	Posição de uma string em outra string	position('om' in 'Thomas')	3
upper(string)	Converte String na sua versão maiúscula	upper('jose')	JOSE
ascii(string)	Retorna o decimal da string	ascii('x')	120
chr(int)	Converte o valor int no caracter correspond	chr (120)	X
initcap(int)	Converte a 1ª. Letra para Maisc.	Initcap ('jose FELIPE')	Jose Felipe
reverse(string)	Acha a string reversa	Reverse('FELIPE')	EPILEF

```
CREATE TABLE PALAVRAS (
        ID INT2,
        PAL_ORIG VARCHAR(20),
        FNC_TRANSF VARCHAR(15),
        RESULT_NUM INT4,
        RESULT_ALFA VARCHAR(20) );


select * from palavras;
--
INSERT INTO PALAVRAS VALUES (1, 'jose', 'bit_length');
--
UPDATE PALAVRAS SET RESULT_NUM = bit_length(PAL_ORIG) WHERE ID = 1;
--
INSERT INTO PALAVRAS VALUES (2, 'jose', 'char_length');
--
UPDATE PALAVRAS SET RESULT_NUM = char_length(PAL_ORIG) WHERE ID = 2;
--
INSERT INTO PALAVRAS VALUES (3, 'JOSE', 'lower');
--
UPDATE PALAVRAS SET RESULT_ALFA = lower(PAL_ORIG) WHERE ID = 3;
--
INSERT INTO PALAVRAS VALUES (4, 'ABACATE', 'position');
--
UPDATE PALAVRAS SET RESULT_NUM = position (PAL_ORIG,'CA') WHERE ID = 4;
--
INSERT INTO PALAVRAS VALUES (5, 'banana', 'upper');
--
UPDATE PALAVRAS SET RESULT_ALFA = upper (PAL_ORIG) WHERE ID = 5;
--
INSERT INTO PALAVRAS VALUES (6, 'B', 'ascii');
--
UPDATE PALAVRAS SET RESULT_NUM = ascii (PAL_ORIG) WHERE ID = 6;
--
INSERT INTO PALAVRAS VALUES (7, '', 'chr',66);
--
UPDATE PALAVRAS SET RESULT_ALFA = chr(RESULT_NUM) WHERE ID = 7;
--
INSERT INTO PALAVRAS VALUES (8, 'jose FELIPE', 'initcap');
--
UPDATE PALAVRAS SET RESULT_ALFA = chr (PAL_ORIG) WHERE ID = 8;
--
INSERT INTO PALAVRAS VALUES (9, 'jose FELIPE', 'reverse');
--
UPDATE PALAVRAS SET RESULT_ALFA = reverse (PAL_ORIG) WHERE ID = 9;
```

2.16. Principais Funções de Data e Hora

Função	Retorno e Descrição	Exemplo	Resultado
age(timestamp)	Interval String	age('1957-06-13')	43 years 8 mons 3 days
current_date	date	current_date	Data Atual
current_time	Time with time zone	current_time	Hora Atual
Localtime	time	localtime	Hora Local

```
CREATE TABLE TEMPORALIDADE (
ID INT2,
IDADE VARCHAR(30),
DATA DATE,
DATA_ATUAL DATE,
HORA_ATUAL TIME,
HORA_LOCAL  TIME);

select * from TEMPORALIDADE;
--
INSERT INTO TEMPORALIDADE VALUES (1, '','1989-01-01');
--
UPDATE TEMPORALIDADE SET IDADE = age(DATA) WHERE ID = 1;
--
-- atribui ao campo DATA_ATUAL a data armazenada no Sistema Operacional
UPDATE TEMPORALIDADE SET DATA_ATUAL = current_date WHERE ID = 1;
--
-- Acrescenta a Data Atual 07 dias a frente
UPDATE TEMPORALIDADE SET DATA_ATUAL = current_date + 7 WHERE ID = 1;
--
-- Recupera do Sistema Operacional a Hora Atual
UPDATE TEMPORALIDADE SET HORA_ATUAL = current_time WHERE ID = 1;
--
-- Recupera do Sistema Operacional a Hora Atual
UPDATE TEMPORALIDADE SET HORA_LOCAL = localtime WHERE ID = 1;
--
```