

POSTGRESQL: O GUIA COMPLETO



by: BINÁRIO
LIVRE

Sumário

Capítulo 1. Conceitos básicos.....	1
1.1 Introdução aos bancos de dados relacionais.....	1
1.1.1 Tabelas.....	1
1.2 Introdução ao SQL.....	1
1.2.1 <i>Data Definition Language</i>	1
1.2.2 Data Manipulation Language.....	2
1.2.3 Data Control Language.....	2
1.2.4 Data Query Language.....	2
Capítulo 2. Tutorial básico.....	3
2.1 Criação e deleção de bancos de dados.....	3
2.2 Acesso ao banco de dados.....	3
2.3 Gerenciamento de tabelas.....	3
2.4 Criação e deleção de tabelas.....	4
2.5 Preenchimento de tabelas.....	4
2.6 Consultas ao banco de dados.....	5
2.6.1 Modificador WHERE.....	6
2.6.2 O modificador DISTINCT.....	6
2.6.3 Funções de agregação.....	6
2.6.4 junções SQL.....	6
2.7 Alteração de linhas.....	7
2.8 Deleção de linhas.....	7
Capítulo 3. Tutorial avançado.....	9
3.1 Views.....	9
3.2 Chaves estrangeiras.....	9
3.3 Transações.....	9
3.4 Window functions.....	10
3.5 Herança.....	10
Capítulo 4. A linguagem SQL.....	12
4.1 Estrutura léxica.....	12
4.2 Identificadores e palavras-chave.....	12
4.3 Constantes.....	12
4.3.1 Strings.....	12
4.3.2 Bit strings.....	13
4.3.3 Constantes numérica.....	13
4.3.4 Operadores.....	13
4.4 Caracteres especiais.....	13
4.5 Comentários.....	14
4.6 Precedência de operadores.....	14
4.7 Casts de tipo.....	15
4.8 Expressões COLLATION.....	15
Capítulo 5. Comandos DDL.....	16
5.1 Valores padrão (default).....	16
5.2 Constraints.....	16
5.2.1 Constraint CHECK.....	16
5.2.2 Constraint NOT-NULL.....	17
5.2.3 Constraint UNIQUE.....	17

5.2.4 PRIMARY KEY.....	17
5.2.5 FOREIGN KEY.....	17
5.3 Colunas do sistema.....	19
5.4 Modificação de tabelas.....	19
5.4.1 Adicionando Colunas.....	19
5.4.2 Removendo Colunas.....	19
5.4.3 Adicionando <i>Constraints</i>	20
5.4.4 Removendo <i>Constraints</i>	20
5.4.5 Alterando tipo de dados de colunas.....	20
5.4.6 Renomeando tabelas e colunas.....	20
5.5 Privilégios.....	21
5.6 Políticas de segurança de linha.....	21
5.7 Esquemas.....	21
5.7.1 Criação de esquemas.....	22
5.7.2 Acessando esquemas.....	22
5.7.2 Criando tabelas em esquemas.....	22
5.7.3 Excluindo esquemas.....	22
5.7.4 O esquema public.....	22
5.7.5 O caminho de busca de esquemas.....	23
5.7.5.1 Alterar o caminho de busca de esquemas.....	23
5.7.6 Esquemas e privilégios.....	23
5.7.7 O Esquema catálogo do sistema.....	24
5.7.8 Padrões de uso de esquemas.....	24
5.8 Portabilidade.....	25
5.9 Herança.....	25
5.10 Exceções.....	27
5.11 Particionamento de tabelas.....	28
5.11.1 Particionamento por intervalos.....	28
5.11.2 Particionamento por lista.....	28
5.11.3 Particionamento declarativo.....	29
5.11.4 Tutorial particionamento.....	30
5.11.5 Manutenção de partições.....	31
5.11.6 Limitações.....	32
5.11.7 Implementação utilizando herança.....	32
5.11.7.1 Tutorial de implementação utilizando herança.....	33
5.11.7.2 Manutenção de partições.....	36
5.11.7.3 Limitações.....	36
5.11.8 Particionamento e <i>constraint exclusion</i>	37
5.11.9 Boas práticas do particionamento declarativo.....	38
5.12 Dados estrangeiros.....	38
5.13 Rastreamento de dependências.....	38
Capítulo 6. Comandos DML.....	40
6.1 Inserindo dados.....	40
6.2 Atualizando dados.....	41
6.2 Excluindo dados.....	42
6.3 Obtendo dados de linhas modificadas.....	42
Capítulo 7. Comando DQL.....	44
7.1 Conceitos gerais.....	44
7.2 Expressões de tabela.....	44
7.3 A palavra-chave FROM.....	45

7.4	Junções entre tabelas.....	45
7.4.1	CROSS-JOIN.....	46
7.4.2	Junções qualificadas.....	46
7.5	Apelidos de tabelas e colunas.....	47
7.6	Sub-consultas.....	48
7.7	Funções de tabela.....	48
7.8	Sub-consultas laterais.....	50
7.9	A cláusula WHERE.....	50
7.10	As cláusulas GROUP BY e HAVING.....	51
7.11	GROUPING SETS, CUBE e ROLLUP.....	54
7.12	Listas de seleção.....	56
7.12.1	Itens da lista de seleção.....	56
7.13	Rótulos de colunas.....	56
7.14	A palavra chave DISTINCT.....	57
7.15	Combinação de consultas.....	57
7.16	Ordenando linhas.....	57
7.17	LIMIT e OFFSET.....	58
7.18	Lista VALUES.....	59
Capítulo 8. Tipos de dados.....		61
8.1	Numéricos.....	61
8.2	Caracteres.....	62
8.3	Data e tempo.....	62
8.3	Booleano.....	62
8.4	Demais tipos.....	63

Capítulo 1. Conceitos básicos

1.1 Introdução aos bancos de dados relacionais

É imprescindível para o início dos estudos da linguagem SQL o conhecimento das entidades e objetos que formam um banco de dados, assim sendo, a intenção destes capítulos iniciais é discorrer sobre os conceitos básicos, e fundamentais, de um banco de dados.

1.1.2 Tabelas

A estrutura inicial de um banco de dados são as tabelas, elas são o principal objeto presente no mesmo, e é o elemento que, de fato, armazena os dados. Há diferentes tipos de bancos de dados, porém o tipo abordado nesta apostila é o relacional, que armazenam relações, ou tabelas, e estas são formadas por colunas (campos) e linhas (tuplas).

1.2 Introdução ao SQL

Esta apostila visa introduzir o estudante a ciência de dados de forma básica a intermediária, apresentando uma didática simples afim tornar a abordagem amigável até a estudantes sem nenhum conhecimento anterior sobre o tema. Portanto este primeiro capítulo irá apresentar conceitos e definições básicas sobre a linguagem SQL e bancos de dados relacionais.

A linguagem SQL é a sintaxe padrão para o gerenciamento de bancos de dados relacionais, contendo palavras-chave, cláusulas e comandos para realizar alterações em um banco de dados relacional de forma concisa, seu nome é um acrônimo para “*Structured Query Language*” ou linguagem de consulta estruturada em tradução livre. O SQL contém diversos tipos de comandos que são comumente categorizados em quatro tipos, sobre os quais discorreremos brevemente a seguir, não se preocupe em entender todas as funções e utilidades de cada comando, a intenção dos tópicos a seguir é somente introduzir os comandos existentes na linguagem SQL, posteriormente retornaremos a estes comandos para estudos aprofundados sobre suas aplicações.

1.2.1 *Data Definition Language*

Os comandos DDL ou “*Data Definition Language*” tem como principal função formar a estrutura das tabelas, ou seja definir a forma em que os dados serão armazenados nas tabelas. Os comandos DDL são:

CREATE: Como o nome sugere, este comando é utilizado para criar objetos no banco de dados como tabelas, colunas, etc...

ALTER: Uma vez criados com o comando CREATE os objetos no banco de dados têm diversas características, como uma tabela tem certo número de colunas, e estas colunas por sua vez armazenam algum tipo de dado, e para alterar tanto a quantidade de colunas de uma tabela quanto o tipo de dado de uma coluna é utilizado o comando ALTER.

DROP: A função do comando DROP é o inverso do comando CREATE, ou seja, ele exclui objetos dos bancos de dados relacionais, como tabelas e colunas.

1.2.2 Data Manipulation Language

Os comandos DML ou “*Data Manipulation Language*” tem como função manipular os dados presentes dentro das tabelas de um banco de dados relacional. Os comandos DML são:

INSERT: Insere dados dentro de uma tabela, preenchendo assim uma ou mais linhas.

UPDATE: Atualiza dados já existentes de uma ou mais linhas.

DELETE: Exclui dados de uma ou mais linhas de uma tabela.

1.2.3 Data Control Language

Os comandos DCL ou “*Data Control Language*” determinam as regras e privilégios dos usuários a determinado banco de dados, esquema ou tabela. Os comandos DCL são:

GRANT: Dá privilégios de alteração a determinado usuário ou grupo de usuários no banco de dados, esquema ou tabela.

REVOKE: Retira privilégios de alteração de determinado usuário ou grupo de usuários no banco de dados, esquema ou tabela.

1.2.4 Data Query Language

O comandos DQL ou “*Data Query Language*” tem como função a realização de consultas a registros de tabelas presentes no banco de dados. Existe somente um comando nesta categoria, conforme visto a seguir:

SELECT: Realiza buscas a registros dentro de uma ou mais tabelas existentes em um banco de dados.

Capítulo 2. Tutorial básico

Tendo conhecimento sobre os principais objetos e comandos de um banco de dados já é hora de iniciar os estudos práticos de alguns comandos fundamentais do gerenciamento de banco de dados.

2.1 Criação e deleção de bancos de dados

Os bancos de dados, como informado no capítulo interior servem para armazenar as tabelas, esquemas entre outras entidades e objetos, então, por consequência lógica devemos iniciar os estudos com a criação de um banco de dados inicial, a sintaxe usada para a criação de um banco de dados é a seguinte:

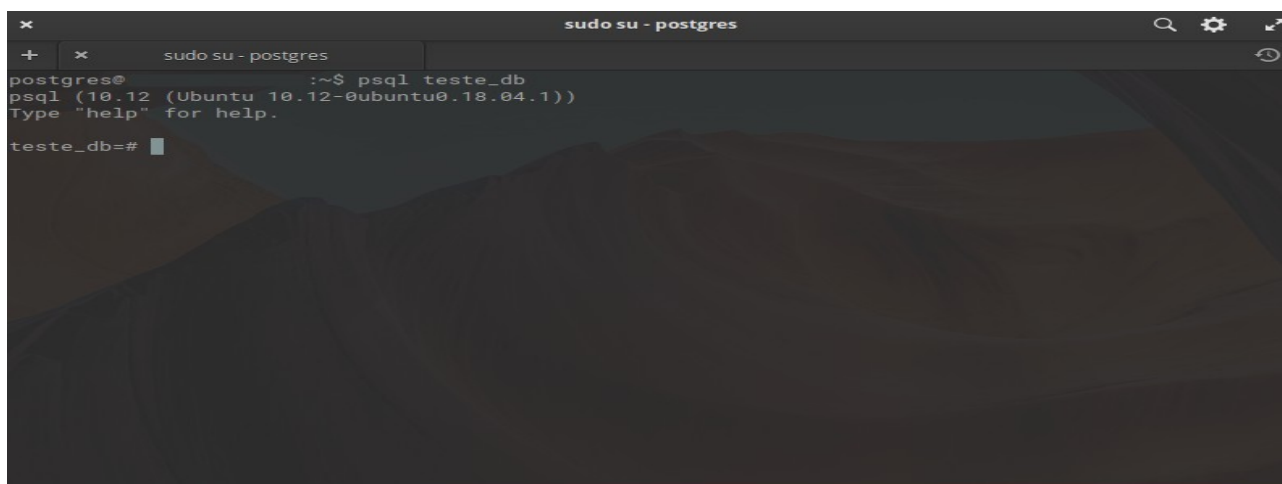
```
=#CREATE DATABASE nome_banco;
```

Já o processo inverso, para a remoção de um banco de dados é realizado pelo comando DROP, conforme o exemplo a seguir:

```
=#DROP DATABASE nome_banco;
```

2.2 Acesso ao banco de dados

O procedimento para o acesso ao banco de dados é diferente a depender de qual DBMS está sendo utilizado. Por exemplo, o psql, um DBMS que utiliza a interface do terminal para receber comandos, utiliza a sintaxe ilustrada abaixo para o acesso à bancos de dados.



```
postgres@ ~$ sudo su - postgres
postgres@ ~$ psql teste_db
psql (10.12 (Ubuntu 10.12-0ubuntu0.18.04.1))
Type "help" for help.

teste_db=#
```

*PARA INFORMAÇÕES SOBRE OUTROS DBMS CONSULTE SUA RESPECTIVA DOCUMENTAÇÃO

2.3 Gerenciamento de tabelas

Este tópico visa apresentar um tutorial introdutório para algumas das funções mais importantes do gerenciamento de tabelas, não é a intenção apresentar profundamente todas as possibilidades de alterações e comandos que podem ser utilizados para realizar

tais funções, algumas definições podem parecer confusas a princípio porém todas serão retomadas em capítulos mais adiante.

2.4 Criação e deleção de tabelas

Após o banco de dados devidamente criado e acessado já é possível criar as tabelas dentro de seu escopo, as tabelas, assim como os bancos de dados, são criadas utilizando o comando CREATE, na sintaxe exemplificada abaixo:

```
=# CREATE TABLE nometabela (  
    nome_coluna1          tipo_coluna1,  
    nome_coluna2          tipo_coluna2  
);
```

Sendo nome_coluna1 e nome_coluna2 dois nomes válidos, ou seja, que não conflitem com nenhuma coluna do sistema postgresql ou palavra-chave da linguagem SQL, ambos serão discutidos posteriormente. E também tipo_coluna1 e tipo_coluna2 dois tipos de dados válidos pelo sistema postgresql, os tipos de dados têm um capítulo específico no final desta apostila, porém alguns dos mais comuns serão apresentados no decorrer da apostila.

O postgresql não é case sensitive para nomes de tabelas ou colunas, portanto o comando a seguir levantaria um erro devido a existência de duas colunas com o mesmo nome:

```
=# CREATE TABLE funcionarios (  
    nome          varchar(20),  
    NOME          integer  
);
```

Há uma forma de sobrepor esta limitação com o uso de aspas duplas (""") tornando o nome case sensitive, portanto, o comando a seguir não resultaria em erro:

```
=# CREATE TABLE funcionarios (  
    nome          varchar(20),  
    "NOME"        integer  
);
```

A sintaxe utilizada para deletar uma tabela é a seguinte:

```
=# DROP TABLE nometabela;
```

2.5 Preenchimento de tabelas

O preenchimento de tabelas é feito pelo comando INSERT, como no exemplo a seguir:

=# INSERT INTO nometabela

VALUES (valor1, valor2);

Mas a seguinte sintaxe é recomendada já que o comando acima insere os dados nas colunas na ordem em que elas foram declaradas na criação da tabela e com o passar do tempo tende a ser difícil se lembrar da ordem exata das colunas.

=# INSERT INTO nometabela (nome_coluna1, nome_coluna2)

VALUES (valor1, valor2);

Dessa forma também é possível omitir colunas, por exemplo, o comando a seguir não apresenta nenhum erro:

CREATE TABLE vendas(

id_produto varchar(20),

valor integer,

quantidade integer,

dia date

);

INSERT INTO vendas (id_produto, valor, quantidade)

VALUES('parabrisa', 200, 2);

Mesmo sem o valor para a coluna dia, o comando INSERT funciona perfeitamente, porém sempre que uma coluna é ignorada ela recebe o valor NULL, que representa nulo, lembrando que nulo é DIFERENTE de zero.

2.6 Consultas ao banco de dados

As consultas ao banco de dados são feitas pelo comando SELECT de acordo com a sintaxe básica escrita abaixo:

SELECT nomecoluna FROM nometabela;

Também são permitidas expressões apresentadas como colunas no comando SELECT como no exemplo abaixo:

SELECT cidade, (maxtemp+mintemp)/2 AS temp_media, data FROM clima;

A requisição acima retorna as colunas cidade, data e temp_media que é igual as colunas maxtemp e mintemp somadas e divididas por 2.

O modificador AS confere um novo rótulo para a coluna de retorno, e pode ser usado em qualquer coluna da seleção.

2.6.1 Modificador WHERE

Outro modificador de consulta é a cláusula WHERE, que especifica quais as linhas que serão exibidas. A sintaxe da cláusula WHERE é a seguinte:

WHERE expressão_booleana;

Ex: **SELECT * FROM clima**

WHERE maxtemp > 35.0;

O simbolo “” significa que todas as colunas devem ser exibidas.

Há também o modificador ORDER BY, que ordena o retorno da requisição de acordo com uma coluna especificada. A sintaxe do modificador ORDER BY é:

ORDER BY nomecoluna;

Ex: **SELECT * FROM clima**

ORDER BY mintemp;

2.6.2 O modificador DISTINCT

O modificador DISTINCT seleciona somente linhas que não se repetem. E é utilizado seguindo a sintaxe apresentada a seguir:

DISTINCT nomecoluna;

Ex: **SELECT DISTINCT mintemp**

FROM clima

ORDER BY mintemp;

2.6.3 Funções de agregação

O postgresql suporta também funções de agregação para computação de uma colunas de várias linhas, algumas das mais utilizadas são: min, max, avg, sum e count. Elas funcionam da seguinte maneira:

nome_funcao(nomecoluna ...)

2.6.4 junções SQL

Quando é necessário fazer consultas a mais de uma tabela ao mesmo tempo é necessário o uso das junções SQL. Para realizar junções entre as tabelas deve-se ter pelo menos uma chave que identifique cada elemento específico presente nas duas colunas, essa chave é, normalmente, ou o nome ou o ID de um determinado dado.

SELECT nomecoluna ...
FROM nometabela1

nomejunção nometabela2

ON (nometabela1.nomechave1 = nometabela2.nomechave2);

Sendo nomecoluna1 e nomecoluna2 duas colunas correspondentes.

Também há a possibilidade de se fazer uma junção de tabela com si mesma (self join) quando é necessária a consulta e comparação entre dados de uma mesma tabela.

Existem alguns tipos de junções como OUTER JOIN e LEFT JOIN, as junções serão abordadas de uma forma mais profunda em capítulos futuros, porém abaixo há uma representação visual.



2.7 Alteração de linhas

Quando necessário atualizar linhas nas tabelas é utilizado o comando UPDATE, que funciona de acordo com a seguinte sintaxe:

UPDATE nometabela

SET nomecoluna = novovalor

WHERE expressão_booleana;

Ex: UPDATE cidades

SET habitantes = 12180000

WHERE cidade = 'São Paulo';

2.8 Deleção de linhas

As deleções de linhas podem ser feitas utilizando o comando DELETE com a seguinte sintaxe:

DELETE FROM nometabela WHERE expressão_booleana;

A cláusula WHERE é de suma importância no comando DELETE pois, sem ela o comando DELETE irá excluir todas as linhas da tabela sem realizar nenhuma confirmação.

Capítulo 3. Tutorial avançado

Este capítulo apresenta alguns aspectos e funções mais avançadas do gerenciamento de bancos de dados mas, assim como o capítulo anterior, não tem como intenção elucidar totalmente cada função, somente introduzi-las ao leitor de forma prática para uma explicação completa em capítulos futuros.

3.1 Views

Para ocultar detalhes de implementação das consultas em aplicações são utilizadas as views, que nada mais são que aliases(apelidos) para as consultas.

```
CREATE VIEW nome_view AS
    SELECT nomecoluna ...
    FROM nometabela ...;
```

No exemplo acima a consulta realizada pelo comando SELECT poderá ser realizada apresentando somente o nome da view.

```
SELECT * FROM nome_view;
```

3.2 Chaves estrangeiras

Muitas vezes, é necessário utilizar tabelas que se relacionam entre si, e quando um valor de uma coluna depende do valor de uma outra coluna devem ser usadas chaves estrangeiras, que não permitem a adição de um valor em uma tabela sem que ele esteja presente em uma outra tabela. Para utilizar chaves estrangeiras basta utilizar a palavra chave REFERENCES na seguinte sintaxe:

```
CREATE TABLE nometabela1 (
    nomecoluna tipocoluna REFERENCES nometabela2(nomecoluna2)
);
```

3.3 Transações

O modelo de bancos de dados transacionais são essenciais para o uso dos bancos de dados relacionais com toda a sua funcionalidade. Por exemplo, se o banco de dados de um banco ao realizar uma transferência e retirar o dinheiro de uma conta e não depositar na conta destino seria um erro desastroso, e isso seria possível de acontecer caso os comandos de retirada de uma conta e depósito na outra não fossem dependentes entre si.

Transações são um bloco de comandos que devem ser feitos em uma só operação atômica, ou seja, de tudo ou nada. Para realizar transações somente é necessária a adição dos comandos BEGIN E COMMIT respectivamente no início e fim de cada bloco de comandos. Como no exemplo a seguir:

```
Ex: BEGIN;

      UPDATE ...

      COMMIT;
```

Ao realizar transações talvez sejam necessários pontos de volta caso sejam realizados comandos errados ou a existência de erros em algum dos comandos. Para realizar checkpoints para retorno dentro de transações é necessário utilizar a palavra-chave savepoint e para retornar a eles é necessária a utilização da palavra-chave rollback.

```
Ex: BEGIN;

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

      SAVEPOINT meu_ponto_de_salvamento;

      UPDATE conta_corrente SET saldo = saldo + 100.00
      WHERE nome = 'Bob';

      -- uai ... o certo é na conta do Pedro

      ROLLBACK TO meu_ponto_de_salvamento;

      UPDATE conta_corrente SET saldo = saldo + 100.00
      WHERE nome = 'Pedro';

      COMMIT;
```

3.4 Window functions

Quando se é necessário separar os conteúdos de uma requisição baseando-se em categorias definidas se utilizam-se window functions utilizando a palavra-chave OVER

Ex: SELECT nomecoluna OVER (especificação de classe Ex:PARTITION BY)
FROM nometabela;

3.5 Herança

Uma das funcionalidades características dos bancos de dados orientados a objetos é a herança, que permite que uma tabela herde de outra tabela suas características.

Esta funcionalidade é performada utilizando a palavra chave INHERITS na sintaxe descrita abaixo:

```
CREATE TABLE nometabela2(  
    nome_coluna    tipo_coluna  
)  
) INHERITS (nometabela1);
```

A tabela herdeira herda todas as colunas da tabela herdada, porém também pode ter suas colunas próprias, como no seguinte exemplo:

```
CREATE TABLE cidades(  
    nome            varchar(20),  
    habitantes      integer  
);  
CREATE TABLE capitais(  
    estado          char(2)  
)  
)INHERITS(cidades);
```

A tabela capitais terá as colunas nome e habitantes assim como a tabela cidades porém também terá adicionalmente a coluna estado.

Capítulo 4. A linguagem SQL

4.1 Estrutura léxica

A estrutura léxica básica da linguagem SQL é composta por comandos, que por sua vez são compostos por tokens e terminados por um semicólon (;). Os tokens podem ser, por sua vez, palavras-chave (UPDATE, INSERT etc...), constantes ou caracteres especiais.

Há também os comentários, que não são considerados tokens, são considerados um tipo de espaço em branco (Ou seja, podem ser utilizados em qualquer lugar, desde de que seja respeitada a sintaxe). Nos tópicos a seguir abordaremos todos os tipos de tokens e também os comentários.

4.2 Identificadores e palavras-chave

Palavras-chave são os tokens pré especificados pela linguagem SQL para servirem como identificadores de comandos, para informar ao SQL qual comando desejamos executar.

Identificadores, também conhecidos como nomes, são os nomes que damos as nossas constraints, tabelas, colunas , etc...

Tanto identificadores de tabelas quanto palavras-chave não são case sensitive na linguagem SQL, ou seja, os comando iniciados com SELECT ou SeleCT serão executados da mesma forma pelo sistema.

Há uma forma de tornarmos os identificadores case sensitive no SQL, basta colocá-los entre aspas duplas (""), como já exemplificado em no tópico 2.4.

Identificadores devem iniciar com letras, números ou underline(_). E de acordo com a constante NAMEDATALEN-1 seguido pelo SQL o tamanho máximo de identificadores é 63 bytes.

Apesar de não ser norma, convencionou-se o uso dos identificadores em caixa baixa e as palavras chave em caixa alta.

4.3 Constantes

Há três tipos de constantes no postgresql, strings, bit strings e números.

4.3.1 Strings

Strings em SQL são sequências de caracteres entre aspas simples ('). Se quiser adicionar apóstrofes dentro de strings adicione mais duas aspas simples.

Ex:'Lion''s den'.

Duas strings com espaços em branco entre si e pelo menos uma quebra de linha entre elas são concatenadas e válidas como uma só string pelo SQL.

Ex: `SELECT 'foo'`

`'bar';`

e

`SELECT 'foobar';`

são lidas igualmente.

O postgresql também aceita as constantes string com barra invertida (\) no estilo da linguagem de programação C e constantes string com caracteres de escape unicode. Por exemplo, para adicionar apóstrofes seguindo as constantes barra invertida no estilo C usaremos a seguinte sintaxe:

Ex: `'Lion\'s Den'`

4.3.2 Bit strings

Constantes bit strings nada mais são que strings que podem conter somente dígitos binários e devem ser iniciadas com um B maiúsculo ou minúsculo seguidos de caracteres binários entre aspas simples. Também existem bit strings hexadecimais que são iniciadas por um X maiúsculo ou minúsculo seguidos de caracteres hexadecimais entre aspas simples. Cada caractere hexadecimal equivale a 4 caracteres binários.

4.3.3 Constantes numérica

Constantes numéricas são simplesmente números, que podem ter expoentes, neste formato `numero[e][sinal][expoente]` e também números decimais utilizando o ponto decimal, assim constantes como 1.925e-3 e 7.50 são aceitas.

4.3.4 Operadores

Nomes de operadores feitos pelo usuário podem ter até 63 caracteres de tamanho, os seguintes caracteres são permitidos: `+ - * / < > = ~ ! @ # % ^ & | ` ?`

Há duas regras para a nomeação de operadores:

1: `'--'` e `'*'` não podem DE NENHUMA FORMA aparecer no nome de operadores.

2: nomes de operadores não podem terminar com os caracteres `'+'` ou `'-'` a não ser que estejam presentes algum dos seguintes caracteres: `! @ # % ^ & | ` ?`.

4.4 Caracteres especiais

Alguns caracteres não alfabéticos pode executar funções específicas dentro da linguagem SQL. Estes são chamados caracteres especiais, sendo eles:

Cifrão (\$): Pode representar constantes string se caracteres forem cercados de cifrões, se seguido de dígitos pode representar um parâmetro posicional em uma função ou declaração.

Parênteses (()): São utilizadas para reforçar a precedência de operadores.

Colchetes ([]): Usados para selecionar elementos de um array.

Virgulas (,): Usada comumente para separar elementos de uma lista.

Dois pontos (:): É utilizado comumente para selecionar "pedaços" de arrays.

Asteriscos (*): Normalmente é utilizado para selecionar todos os elementos de uma tabela e também utilizado em funções de agregação para sinalizar que a função não recebe parâmetros.

Ponto (.): O ponto é utilizado em numerais como ponto decimal e também para separar nomes de tabelas-colunas.

4.5 Comentários

Em SQL há duas formas diferentes de se fazer comentários: respeitada SINTAXE

1: Utilizando '--'. Ex: --Isso é um comentário SQL

2: Utilizando '/* */', como em C. Ex: /*Este é um comentário SQL*/

A única regra sobre os comentários é que eles não devem interferir na sintaxe SQL, ou seja, não aparecer no meio de tokens da linguagem.

Ex: **SELE--comentárioCT ***

FROM tabela1; *ERRADO

SELECT * --comentário

FROM tabela1; *CERTO

4.6 Precedência de operadores

A precedência da maioria dos operadores no postgresql é a mesma o que torna complicado a previsão da precedência deles pelo parser. É altamente recomendado o uso de parênteses para reforçar a precedência desejada. Na tabela abaixo estão descritos a maioria dos operadores presentes no sistema postgresql:

OPERADOR	ASSOCIATIVIDADE	DESCRIÇÃO
.	Esquerda	Separador de nome tabela/coluna
::	Esquerda	Cast de tipo estilo postgresql
[]	Esquerda	Seletor de elemento de array

+ -	Direita	mais/menos unário
* / %	Esquerda	Multiplicação, divisão e módulo
+ -	Esquerda	Adição/subtração
Outros operadores	Esquerda	Qualquer outro operador nativo ou do usuário
BETWEEN IN LIKE ILIKE	N/A	Comparação de strings
< > = <= >= <>	N/A	Operadores de comparação
IS ISNULL NOTNULL	N/A	IS TRUE, IS FALSE, IS NULL etc..
NOT	Direita	Negação lógica
AND	Esquerda	Conjunção lógica
OR	Esquerda	Disjunção lógica

4.7 Casts de tipo

Os casts de tipo em SQL são uma forma de transformar um tipo de dado em um outro tipo de dado especificado. Os casts seguem duas sintaxes distintas:

CAST (tipo1 AS tipo2)

ou

tipo1::tipo2

4.8 Expressões COLLATION

Uma expressão collation define o tipo de organização e ordenação dos caracteres das string. Caso não haja nenhum tipo de expressão collation o sistema postgresql ordenará os caracteres de acordo com o sua identificação UTF-8. As expressões collation seguem a seguinte sintaxe:

expressão COLLATE tipodeagrupamento;

Capítulo 5. Comandos DDL

Os comandos de definição de dados (DDL) são utilizados para definir aspectos básicos das tabelas, como criá-las, modificá-las, excluí-las. Como visto no tutorial básico os comandos iniciais para tabelas são os de criação e exclusão de tabelas CREATE e DROP, para revisar a sintaxe básica, retorne ao tópico 1.2.1.

5.1 Valores padrão (default)

Pode ser necessário o uso de valores default para colunas quando é desejado que a coluna receba um valor padrão mesmo quando nenhum valor for designado a ela. Os valores default são atribuídos seguindo a seguinte sintaxe:

```
CREATE TABLE nometabela(  
    nomecoluna      tipocoluna DEFAULT valor1  
);
```

Sendo valor1 um valor compatível com o tipo de dados da coluna.

5.2 Constraints

Constraints são basicamente restrições de manipulação para colunas. Por exemplo, é interessante que, em uma coluna que armazena preços de produtos seja restrita a valores positivos. Há diversos tipos de constraints suportados pelo postgresql, entre eles estão os seguintes:

5.2.1 Constraint CHECK

Constraints check são o tipo mais simples de constraints, elas têm uma ampla gama de funcionalidades, e fundamentalmente checam se uma expressão dada é verdadeira ou falsa, se falsa, a constraint check acusa erro. Há dois tipos de constraints check, as constraints check de coluna e as constraints check de tabela, e a diferença entre elas é somente que o escopo das constraints check de tabela abrangem todas as colunas da tabela já as constraints de coluna afetam somente suas respectivas colunas.

As constraints check seguem a seguinte sintaxe:

```
CREATE TABLE nometabela(  
    nomecoluna1      tipocoluna1 CONSTRAINT check1 CHECK  
    (expressão_booleana),  
    CONSTRAINT check2 CHECK (expressão_booleana)  
);
```

O exemplo acima cria uma coluna e as constraints check de nome check1 e check2 que são, respectivamente, constraints check de coluna e de tabela.

5.2.2 Constraint NOT-NULL

A constraint NOT-NULL certifica que uma coluna não pode ter valor nulo, na seguinte sintaxe:

```
CREATE TABLE nometabela(  
    nomereg1 tiporeg1 NOT NULL  
);
```

Há também a constraint NULL, que certifica que a coluna recebe null como valor default, e é equivalente a omitir a constraint NOT NULL, o que torna seu uso não convencional.

5.2.3 Constraint UNIQUE

A constraint unique restringe os valores da coluna a serem únicos, ou seja o valor da coluna não pode se repetir em nenhuma linha da tabela.

```
CREATE TABLE nometabela(  
    product_no integer UNIQUE,  
    price numeric  
);
```

5.2.4 PRIMARY KEY

As constraints *primary key* são restrições normalmente dadas a colunas que são identificadores de suas respectivas colunas, essa restrição é uma junção das constraints NOT-NULL e UNIQUE, portanto as colunas *primary-key* devem ser únicas e não nulas.

Tabelas só podem ter uma primary key. Primary keys tem a seguinte sintaxe:

```
CREATE TABLE nometabela(  
    product_no integer PRIMARY KEY,  
    price numeric  
);
```

5.2.5 FOREIGN KEY

As *foreign keys* firmam uma relação entre duas colunas de tabelas diferentes, de forma que, para que seja adicionado um valor na coluna é necessário que esse valor esteja presente na coluna de referência. As *foreign key* são muito úteis para relacionar diretamente diversas tabelas, e utilizam a seguinte sintaxe:

```

CREATE TABLE nometabela1( /*Tabela referencia*/
    product_no integer PRIMARY KEY,
    price numeric
);

CREATE TABLE nometabela2( /*Tabela referenciadora*/
    product_no integer REFERENCES nometabela1 (product_no),
    order_no integer,
    price numeric
);

```

Foreign keys também podem ser declaradas como constraints de tabela:

```

CREATE TABLE nometabela1( /*Tabela referencia*/
    product_no integer PRIMARY KEY,
    price numeric
);

CREATE TABLE nometabela2( /*Tabela referenciadora*/
    product_no integer,
    order_no integer,
    quantity integer,
    unit_price numeric,
    FOREIGN KEY (product_no, unit_price) REFERENCES nometabela1
    (product_no, price)
);

```

Quando um dado é atualizado ou deletado na tabela de referência isso pode gerar erros na tabela na qual ela é referenciada. Para lidar com essas exceções existem as opções de chave estrangeiras:

ON DELETE RESTRICT: Impede que o registro seja alterado ou deletado na tabela referencia caso ainda haja dados ligados a ela na tabela referenciadora.

ON DELETE CASCADE: Deleta ou atualiza automaticamente os dados na tabela referenciadora caso alguma destas ações seja feita na tabela referencia.

ON DELETE NO ACTION: É semelhante a opção on delete restrict porém realiza uma verificação de integridade após qualquer tentativa de alteração na coluna referencia, é a opção padrão quando nenhuma outra é indicada.

ON DELETE SET NULL:Caso algum valor seja alterado ou deletado na coluna referencia o valor correspondente na tabela referenciadora se tornará NULL.

ON DELETE SET DEFAULT:Define um valor padrão para o registro na tabela referenciadora caso a coluna correspondente na tabela referencia seja alterado ou excluído.

5.3 Colunas do sistema

Algumas colunas são criadas automaticamente pelo sistema. Portanto não podem ser criadas colunas com nomes idênticos pelo usuário, não há motivo para se preocupar com elas mas é uma boa prática o conhecimento da existência dessas colunas. Algumas delas são:

OID, TABLEOID, XMIN, CMIN, XMAX, CMAX, CTID;

5.4 Modificação de tabelas

Para modificar colunas e outras características de tabelas você pode simplesmente excluí-las e adicioná-las novamente, mas isso não é uma opção conveniente, principalmente quando já se tem dados registrados na tabela. Ao invés disso, pode se usar o comando ALTER TABLE.

5.4.1 Adicionando Colunas

ALTER TABLE nometabela ADD COLUMN nomecoluna tipocoluna;

A coluna é preenchida com o valor padrão ou NULL caso não haja valor default. Usando essa abordagem é possível realizar qualquer alteração na tabela. A adição de uma coluna com um valor default requer a atualização de todas as linhas para inserir o valor default da nova coluna, portanto se forem usados em maior partes dados que não sejam o valor default é uma boa prática que se adicione a coluna sem o valor default, se adicione os valores diferentes do default usando o comando INSERT e depois disso se adicione o valor default na coluna.

5.4.2 Removendo Colunas

ALTER TABLE nometabela DROP COLUMN nomecoluna;

Todos os registros da coluna apresentada serão excluídos porém nem todas as referências a ela como *foreign keys* seguirão esse processo automaticamente, para apagar todos os dados correspondentes à coluna utilize a palavra chave CASCADE:

ALTER TABLE nometabela DROP COLUMN nomecoluna CASCADE;

5.4.3 Adicionando *Constraints*

ALTER TABLE nometabela ADD TIPO CONSTRAINT;

A constraint é checada imediatamente após ser adicionada, então os dados da tabela devem satisfazer a constraint antes de sua adição. Caso o contrário o comando apresentará erro.

5.4.4 Removendo *Constraints*

Para remover uma constraint deve-se saber seu nome, se o nome foi adicionado pelo usuário então é só digitá-lo, porém se o nome da constraint não foi explicitado em sua criação o sistema deu um nome automaticamente a ela e o comando \d pode ajudar o usuário a encontrá-la.

ALTER TABLE nomecoluna DROP CONSTRAINT nomeconstraint;

Alterando valores default de colunas:

ALTER TABLE nometabela ALTER COLUMN nomecoluna SET DEFAULT valor1;

Isso somente altera o valor default para adições futuras.

O seguinte comando exclui o valor default e torna o default NULL:

ALTER TABLE nometabela ALTER COLUMN nomecoluna DROP DEFAULT;

5.4.5 Alterando tipo de dados de colunas

ALTER TABLE nometabela ALTER COLUMN nomecoluna TYPE tipodedado;

Esta abordagem altera os valores presentes na coluna para um novo tipo se o cast puder ser feito automaticamente pelo postgresql, caso este não seja o caso a palavra-chave USING deve ser usada para detalhar a conversão desejada. As constraints e valor default da coluna também são convertidos pelo postgresql o que pode gerar resultados inesperados, então é uma boa prática excluir as constraints existentes e criá-las novamente para o novo tipo.

5.4.6 Renomeando tabelas e colunas

O comando ALTER TABLE também permite a renomeação de tabelas e colunas, a sintaxe para ambas modificações é bem parecida sendo a renomeação de tabelas feita da seguinte forma:

ALTER TABLE nometabela RENAME TO novonome;

Já a renomeação de colunas se dá da seguinte forma:

ALTER TABLE nometabela RENAME COLUMN nomecoluna TO novonome;

5.5 Privilégios

Quando uma tabela é criada todos os privilégios são concedidos ao seu criador. Para conceder, ou retirar, a outros usuários privilégios de alteração na tabela usam-se as palavras-chave GRANT E REVOKE.

GRANT TIPOPRIVILÉGIO ON nometabela TO nomeusuario;

REVOKE TIPOPRIVILÉGIO ON nometabela FROM nomeusuario;

Tipos de privilégios:

SELECT	INSERT	UPDATE	DELETE
TEMPORARY	EXECUTE	TRUNCATE	REFERENCES
TRIGGER	CREATE	CONNECT	

5.6 Políticas de segurança de linha

Em adição aos privilégios padrão do sistema SQL o postgresql também aceita as políticas de segurança de linha, que são uma camada de privilégios a parte dos privilégios comuns. Para adicionar políticas de segurança de linha, primeiramente deve-se ativar as políticas de segurança de linha.

ALTER TABLE nometabela ENABLE ROW LEVEL SECURITY;

E após isso, adicionar as políticas:

CREATE POLICY nomepolítica ON nometabela TO nomegrupo

USING (expressão_booleana);

Ou pode-se também utilizar as políticas para somente um tipo de privilégio. Usando a sintaxe a seguir:

CREATE POLICY nomepolítica ON nometabela(nomecoluna)

FOR tipoprivilégio

USING (expressão_booleana);

5.7 Esquemas

Um cluster de bancos de dados contém um ou mais bancos de dados, e os bancos de dados por sua vez comportam um ou mais esquemas, que por sua vez comportam tabelas, tipos de dados, funções e operadores. Os nomes de objetos podem ser repetidos em diferentes esquemas dentro do mesmo banco de dados sem apresentar conflitos. O postgresql permite que os usuários acessem vários esquemas que estejam dentro de seus bancos de dados simultaneamente, desde que tenham privilégios para acessá-los.

Esquemas são semelhantes a pastas dentro de um sistema operacional, com a diferença que não podem ser aninhados.

5.7.1 Criação de esquemas

CREATE SCHEMA nomeesquema;

Caso queira criar um esquema em propriedade de outro usuário a seguinte sintaxe é usada:

CREATE SCHEMA nomeesquema AUTHORIZATION nomeusuario;

5.7.2 Acessando esquemas

Para acessar ou criar objetos dentro de um esquema usa-se a seguinte sintaxe:

nomeesquema.nomeobjeto;

Ou uma abordagem ainda mais generalista:

nomedobanco.nomeesquema.nomeobjeto;

5.7.2 Criando tabelas em esquemas

A sintaxe para se criar tabelas dentro de um esquema específico é muito parecida com a sintaxe comum, com a diferença que deve-se especificar o nome do esquema em que se deseja criar a tabela juntamente com o nome da tabela a ser criada.

```
CREATE TABLE nomeesquema.nometabela (  
           nomecoluna1      tipocoluna1,  
           nomecoluna2      tipocoluna2  
);
```

5.7.3 Excluindo esquemas

Para excluir esquemas é usada a seguinte sintaxe caso o esquema não contenha nenhum objeto:

DROP SCHEMA nomeesquema;

Caso o esquema ainda contenha qualquer tipo de objeto use:

DROP SCHEMA nomeesquema CASCADE;

A adição da palavra-chave **CASCADE** deletará juntamente com o esquema todos os objetos nele presentes.

5.7.4 O esquema *public*

No tutorial básico foi descrita a sintaxe para a criação de tabelas sem especificar o esquema usado, nesse caso o esquema definido é o esquema padrão, o esquema *public*.

Então os seguintes comandos são interpretadas igualmente:

```
CREATE TABLE tabela1 (...);
```

```
CREATE TABLE public.tabela1(...);
```

5.7.5 O caminho de busca de esquemas

Os nomes qualificados não são uma forma muito dinâmica de se referenciar objetos dentro de um banco de dados, e essa sintaxe também não pode ser usadas em aplicações. Por isso as tabelas são normalmente referenciadas pelo seu nome desqualificado, que é somente o nome da tabela, O sistema define que somente os esquemas inclusos no `search_path` podem ser referenciados por seus nomes desqualificados.

Porém ao adicionar um esquema ao `search_path` isso concede a todos os usuários do sistema o privilégio `CREATE`, ou seja, todos os usuários do banco de dados podem criar objetos dentro desse esquema.

Há também outro problema da adição de esquemas ao `search_path`, ao realizar uma pesquisa o sistema assume que o primeiro resultado é a resposta à pesquisa mesmo que haja outros resultados com o mesmo nome posteriormente na ordem de busca de esquemas. Então usuários podem fazer seu banco de dados se comportar de forma inesperada pela criação de diferentes objetos com os mesmos nomes.

5.7.5.1 Alterar o caminho de busca de esquemas

Para alterar o caminho de busca de esquemas ou *search path* primeiro é interessante ver qual o caminho de busca de esquemas atual, e para isso usa-se o seguinte comando:

```
SHOW search_path;
```

Esse comando retornará todos os esquemas inclusos em ordem. O comando de alteração do caminho de busca de esquemas por sua vez é o seguinte:

```
SET search_path TO nomeesquema,public;
```

O primeiro esquema do caminho de busca de esquemas é o padrão quando um novo objeto é criado. O esquema `public` não tem nenhuma característica especial, exceto que existe por padrão, ele pode ser excluído também. Portanto, se for interessante que se evite a busca pelo esquema `public` também poderia se usar a seguinte mudança no caminho de busca de esquemas, omitindo o esquema `public`:

```
SET search_path TO nomeesquema;
```

Então, o esquema `public` só seria acessado com declaração explícita antes do nome do objeto a ser criado, como no exemplo do tópico 5.7.2.

5.7.6 Esquemas e privilégios

Por padrão somente os usuários criadores dos esquemas têm acesso aos objetos presentes neles. Para permitir esse acesso a outros usuários do banco de dados o criador

do esquema deve utilizar o tipo de privilégio USAGE, para demais tipos de direitos outros tipos de privilégios talvez sejam necessários. Um usuário pode também receber direitos de criar objetos em um esquema de propriedade de outro usuário, para permitir esse direito o privilégio CREATE precisa ser concedido ao usuário no esquema. Por padrão todos os usuários conectados no banco de dados tem os privilégios CREATE e USAGE no esquema public, porém há formas de se retirar esses privilégios:

REVOKE CREATE ON SCHEMA public FROM PUBLIC;

5.7.7 O Esquema catálogo do sistema

Além do esquema public há também outro esquema que está presente por padrão no postgresql o pg_catalog, que basicamente é um catálogo de todos os objetos criados por padrão pelo sistema, como tabelas, operadores e funções.

O esquema pg_catalog está sempre presente no caminho de busca de esquemas do postgresql. Mas o usuário pode adicioná-lo no final do search_path afim de dar preferência a seus objetos sobre os do sistema durante a busca.

As tabelas do sistemas têm por convenção nomes iniciados em "pg_", portanto essa sintaxe para nomeação de tabelas deve ser evitada pelos usuário ao se criar tabelas mesmo que não exista tabelas do sistema com o respectivo nome para evitar problemas de compatibilidade com atualizações futuras do postgresql.

5.7.8 Padrões de uso de esquemas

Há diversos modos de se organizar seus arquivos por meio da utilização de esquemas. Os padrões de segurança de uso de esquemas previnem que usuários maliciosos possam realizar ações dentro do banco de dados.

Se um banco de dados não usa os padrões de segurança, o usuário que quiser garantir a integridade de seus dados deve iniciar sua sessão alterando o caminho de busca de esquemas para uma string vazia ou remover esquemas não alteráveis pelo superuser.

Há padrões de segurança de esquemas suportados pelo padrão do sistema postgresql como os seguintes:

1: Um padrão de segurança para uso de esquemas é a retirada do direito de criação no esquema public de todos os usuários no esquema public com o comando

REVOKE CREATE ON SCHEMA public FROM PUBLIC

e, em seguida, criar esquemas com os nomes de todos os usuários e alterar o search_path para iniciar as buscas em \$user. O que faz com que cada usuário utilize o seu próprio esquema por padrão.

Após isso se necessário pode se fazer buscas por objetos com os mesmos nomes de objetos presentes no esquema pg_catalog no esquema public.

Esse padrão de segurança só é confiável caso o usuário malicioso **NÃO TENHA** o privilégio CREATEROLE, nesse caso não há padrão de segurança possível.

2: Remover o esquema public do caminho de buscas de esquemas modificando o arquivo postgresql.conf ou pelo comando

ALTER ROLE ALL SET search_path = "\$user".

Neste caso, todos os usuários têm direitos de criação de objetos no esquema public porém só usuários permitidos poderão acessá-los.

Chamadas a funções no esquema public poderão apresentar resultados inesperados, portanto se o banco de dados faz uso de chamadas para funções é recomendada a utilização do padrão anterior.

3: Manter o padrão, garantindo a todos os usuários o acesso ao esquema public, só é recomendado quando não há a possibilidade do uso de esquemas ou quando o banco de dados tem somente um usuário ou uma pequena quantidade de usuários confiáveis.

Para qualquer um dos padrões acima, quando necessária a instalação de plugins ou tabelas de terceiros deve-se inseri-los em um esquema separado e que todos os usuários tenham privilégios para que seja possível realizar consultas e acessos aos objetos presentes nesse respectivo esquema.

5.8 Portabilidade

Segundo o padrão SQL a noção de objetos em um mesmo esquema serem propriedade de diferentes usuários não existe, assim como o conceito do esquema public.

Algumas implementações utilizam apenas um esquema com o nome do usuário, não permitindo qualquer outra sintaxe para a nomeação de esquemas.

Portanto, se desejada a máxima compatibilidade com as demais implementações e também com o padrão SQL deve se evitar a implementação de esquemas de forma complexa. Até devido a algumas implementações não utilizarem esquemas de nenhuma forma.

5.9 Herança

No postgresql uma tabela pode herdar zero ou mais tabelas, e uma consulta pode referenciar tanto todas as linhas de uma tabela quanto todas as linhas de uma coluna e suas descendentes, o que é o padrão. Para fazer uma consulta ao banco de dados com somente a tabela especificada deve-se utilizar a palavra-chave ONLY.

SELECT * FROM ONLY tabela1;

No exemplo acima o escopo da consulta seria somente a tabela1 e nenhuma das tabelas abaixo dela na hierarquia de herança. Vários comandos já citados acima como UPDATE E DELETE também suportam a palavra-chave ONLY.

Há também outra sintaxe para explicitar que as tabelas-filhas incorporarão os escopos da requisição, a adição de asteriscos(*)).

```
SELECT * FROM *tabela1;
```

Essa sintaxe não é obrigatória, é uma herança de versões anteriores do postgresql.

A herança não inclui automaticamente as tabelas-filhas no escopo dos comando INSERT e COPY.

```
EX:CREATE TABLE casa (  
           numero      integer,  
           comodos    smallint,  
  
);  
  
CREATE TABLE predio (  
           andares smallint  
  
) INHERITS (casa);
```

O seguinte comando:

```
INSERT INTO casa (numero, comodos, andares)  
  
VALUES (222, 4, 1);
```

não funcionaria porque o comando INSERT insere dados em uma, e só em uma, tabela nunca em suas herdeiras.

Todas as constraints CHECK e NOT-NULL na tabela herdada são passados para a tabela herdeira, exceto aqueles sinalizados como não herdáveis com a palavra-chave NO INHERIT, todas as outras constraints não são herdadas por padrão.

Uma tabela pode herdar mais de uma tabela. Neste caso a junção entre as duas tabelas herdadas é passada para a tabela herdeira junto com as colunas específicas da tabela herdeira. Caso haja colunas com o mesmo nome em diversas tabelas herdadas ou em tabelas herdadas e na declaração da tabela herdeira essas tabelas serão "unidas", para serem unidas elas devem ser do mesmo tipo de dado, caso o contrário um erro ocorrerá.

Para constraints CHECK e NOT NULL seguem o mesmo padrão. Por exemplo, uma coluna herdada vai ser marcada como NOT NULL se qualquer uma das colunas herdadas estiver marcada como NOT NULL. Constraints CHECK vão ser unidas somente se elas tiverem o mesmo nome, e resultarão em erro caso suas condições sejam conflitantes.

O modo mais comum de realizar herança entre tabelas é durante a criação da tabela utilizando a palavra-chave INHERITS junto ao comando CREATE TABLE como mostrado

no exemplo acima. Porém há outra maneira de realizar a herança quando a tabela já foi criada, utilizando a palavra-chave INHERIT com o comando ALTER TABLE:

ALTER TABLE tabela_herdeira INHERITS tabela_herdada;

Porém para a operação não apresentar erros a tabela herdeira deve conter todas as colunas da tabela herdada e também as constantes CHECK com mesmos nomes e expressões. Essa opção pode ser útil quando é necessário realizar o particionamento de tabelas utilizando herança, que será introduzido posteriormente.

Uma maneira dinâmica de criar tabelas compatíveis a se tornarem tabelas herdeiras posteriormente é utilizar a palavra-chave LIKE na criação da tabela, se a tabela a ser herdada tiver constraints check deve-se incluir a palavra-chave INCLUDING CONSTRAINTS a palavra-chave LIKE.

Uma tabela herdada não pode ser excluída enquanto existirem tabelas herdeiras. Assim como colunas e constraints check de colunas herdeiras não podem ser alteradas ou excluídas se são herança de outras tabelas, uma opção para excluir uma tabela e todas as suas herdeiras de maneira simples é utilizar a palavra-chave CASCADE, esse método também pode ser utilizado no comando ALTER TABLE.

DROP TABLE tabela_herdada CASCADE;

Consultas checam a permissão de acesso somente a tabela herdada. Portanto, permitindo o acesso a tabela herdada garante diretamente o acesso a tabela herdeira, essa característica tem a intenção de manter a associatividade dos dados presentes na tabela herdeira com tabela herdada. Porém a tabela herdeira não pode ser acessada diretamente sem a checagem de privilégios.

Há exceções para essa regra que são os privilégios TRUNCATE e LOCK TABLE, em que os privilégios na tabela herdeira sempre serão checados.

De maneira parecida, as políticas de segurança de tabelas são aplicada somente quando a consulta é explícita somente a tabela herdeira.

Tabelas estrangeiras também podem fazer parte das hierarquias de herança, assim como as tabelas herdeiras e herdadas a tabelas estrangeiras também têm suas operações limitando toda a hierarquia (se uma operação não é aceita na tabela estrangeira não será aceita em nenhuma tabela do sistema hierárquico de herança).

5.10 Exceções

O uso de hierarquias de herança entre tabelas deve ser cuidadoso devido ao fato de que nem todos os comandos SQL funcionarão de forma esperadas nelas.

No caso dos comandos usados para consulta de dados, modificação de dados ou modificação de esquemas como SELECT e UPDATE se propagarão para as tabelas herdeiras por padrão. Comandos que fazem manutenção de banco de dados como

REINDEX e VACUUM normalmente trabalham somente em cada tabela individual, não suportando propagação hierárquica.

Uma das principais limitações das hierarquias de heranças é a **NÃO** propagação de algumas constraints como UNIQUE e chaves estrangeiras só se aplicam a tabelas individuais.

Como, por exemplo, Se declaramos uma coluna como PRIMARY KEY na tabela herdada, isso não impede que esta contenha esta coluna repetida devido a presença de valores repetidos em uma tabela herdeira. E estes valores serão mostrados por padrão em requisições usando SELECT sem o uso da palavra-chave ONLY.

Caso adicione uma *foreign key* utilizando REFERENCES na tabela herdada isso não se propagará automaticamente para as tabelas herdeiras. Isso pode ser superado adicionando a mesma relação de chave estrangeira na tabela herdeira.

Caso uma coluna de outra tabela referencie uma coluna da tabela herdada ela não terá acesso aos dados da mesma coluna na tabela herdeira, e não há uma forma de superar essa limitação.

5.11 Particionamento de tabelas

O sistema do postgresql permite o uso da funcionalidade de particionamento de tabelas. O particionamento de tabelas é, basicamente, o fracionamento dos dados em tabelas menores ao invés da utilização de uma tabela maior para tal função.

A utilização deste tipo de alternativa oferece diversos benefícios como o aumento da velocidade de acesso aos dados e o aumento da performance durante a manipulação destes dados.

Os benefícios do particionamento de tabelas somente são alcançados quando o armazenamento dos dados de forma convencional resultaria em uma tabela muito grande.

O postgresql suporta duas formas de particionamento, sendo elas:

5.11.1 Particionamento por intervalos

A tabela é particionada em "intervalos" ou "*ranges*" definido por uma coluna chave ou conjunto de colunas, sem conflitos entre os intervalos de cada partição como, por exemplo, uma partição pode ser definida por intervalos de datas ou de ID's.

5.11.2 Particionamento por lista

A tabela é particionada listando explicitamente quais valores-chave aparecem em cada partição.

Alguns usuários podem preferir utilizar outras formas de particionamento de tabelas como por meio de herança ou a palavra-chave UNION ALL. Estes métodos oferecem maior

flexibilidade porém não oferecem os mesmos benefícios dos modos de particionamento declarativos oferecidos nativamente pelo postgresql.

5.11.3 Particionamento declarativo

Postgresql oferece uma forma de especificar qual o método de divisão da tabela. Esse método consiste na forma de particionamento e uma lista de colunas ou expressões a serem usadas como chaves de partição.

Todas as linhas inseridas em uma tabela particionada serão redirecionadas para uma partição da mesma baseada no valor da chave de partição. Cada partição têm conjuntos de dados definidos por seus limites de partição. Os métodos de particionamento suportados atualmente são por intervalo e lista de chaves.

As partições das tabelas também podem ser particionadas e receber constraints e índices diferentes entre si.

É impossível transformar uma tabela comum em uma tabela particionada e vice-versa, mas é possível conectar uma tabela comum como uma partição de uma tabela particionada, e é possível desassociar uma partição de uma tabela particionada, tornando-a assim uma tabela comum. Essa operação é realizada utilizando as palavras-chave ATTACH e DETTACH.

As partições são ligadas a tabela particionada por meio de herança. Porém as tabelas particionadas e partições não podem ter nenhum laço de herança com tabelas normais, uma partição não pode herdar nenhuma outra tabela que não a tabela da qual ela é uma partição, nem tanto uma tabela particionada pode ser uma tabela herdada de outra tabela que não suas partições.

As demais funcionalidades das hierarquias de herança funcionam perfeitamente nas tabelas particionadas com algumas ressalvas:

As constraints CHECK e NOT NULL são sempre são herdadas por todas suas partições. Constraints CHECK com a palavra-chave NO INHERIT não são permitidas em tabelas particionadas.

O uso da palavra-chave ONLY para adicionar e excluir constraints somente da tabela particionada funcionará somente caso não haja partições, caso contrário retornará um erro. A adição e exclusão de constraints que estão presentes somente nas partições e não na tabela particionada são permitidos. O uso do comando TRUNCATE em partições não é permitido.

Partições não podem conter colunas que não estão presentes na tabela particionada, também não é possível especificar colunas quando criando partições usando CREATE TABLE, nem tanto adicionar colunas após sua criação por meio do comando ALTER TABLE.

Não é permitida a exclusão da constraint NOT NULL de uma partição caso ela também esteja presente na tabela particionada.

Colunas de partições podem ser chaves estrangeiras de outras tabelas, mas com algumas limitações.

5.11.4 Tutorial particionamento

1: criar a tabela particionada utilizando a palavra chave PARTITION BY utilizando algum método de particionamento, neste caso intervalo:

```
CREATE TABLE medicao (  
  city_id      int NOT NULL,  
  data         date NOT NULL,  
  temperatura int  
) PARTITION BY RANGE (data);
```

2: criar as partições desejadas utilizando a palavra chave PARTITION OF e especificando os valores limite de cada partição:

```
CREATE TABLE medicao_2010fev PARTITION OF medicao  
FOR VALUES FROM ('2010-02-01') TO ('2010-03-01');  
CREATE TABLE medicao_2010mar PARTITION OF medicao  
FOR VALUES FROM ('2010-03-01') TO ('2010-04-01');  
CREATE TABLE medicao_2010abr PARTITION OF medicao  
FOR VALUES FROM ('2010-04-01') TO ('2010-05-01');
```

Pode-se utilizar um segundo método de particionamento a nível de partição:

```
CREATE TABLE medicao_2010abr PARTITION OF medicao  
FOR VALUES FROM (20) TO (35)  
PARTITION BY RANGE (temperatura);
```

Cuidado ao adicionar métodos de particionamento a nível de partição para que esses métodos não ultrapassem os limites da partição a qual eles pertencem assim levando o sistema a armazenar dados de forma inesperada, o sistema não checa esse tipo de erro.

3: Criar um índice nas colunas chave, assim como índices em cada partição:

```
CREATE INDEX ON medicao_2010fev (data);  
CREATE INDEX ON medicao_2010mar (data);  
CREATE INDEX ON medicao_2010abr (data);
```

A criação de índices de chaves não é obrigatório mas é recomendado e pode ser útil na maioria dos casos. Se desejado que cada valor-chave seja único então deve ser criado sempre uma constraint UNIQUE ou PRIMARY KEY para cada partição.

4: Esteja certo de que o parâmetro `constraint_exclusion` esteja habilitado em `postgresql.conf`. Caso esteja desabilitado as requisições não serão otimizadas como desejado. **Atenção!** O parâmetro `constraint_exclusion` deve estar com o valor **ON**, o valor padrão `PARTITION` não habilita totalmente a função.

5.11.5 Manutenção de partições

Normalmente o conjunto de partições criadas na definição inicial da tabela particionada não permanecerão estáticas. É uma prática comum querer remover partições antigas e periodicamente adicionar novas partições para novos dados.

Uma vantagem mais importante do particionamento é justamente a praticidade e segurança maiores de se manipular dados em porções menores.

A maneira mais simples de se remover dados antigos é excluir a partição que não é mais necessária:

```
DROP TABLE medicao_2010fev;
```

Isso pode ser uma forma muito rápida de deletar milhões de dados porque não há a necessidade de deletar todas as gravações individualmente. O comando acima requer o privilégio `ACCESS EXCLUSIVE` na tabela particionada.

Outra forma de realizar a remoção de uma partição é desvincular a partição da tabela particionada, mas manter acesso a partição como uma tabela comum:

```
ALTER TABLE medicao DETACH PARTITION medicao_2010fev;
```

Isso permite operações futuras com os dados que na forma anterior seriam excluídos. Por exemplo, após essa operação é possível realizar um backup dos dados usando `COPY`, `pg_dump`, ou outras ferramentas.

De forma parecida podemos adicionar novas partições para armazenar dados novos.

Pode-se criar novas partições de forma igual as partições anteriores:

```
CREATE TABLE medicao_2010mai PARTITION OF medicao  
FOR VALUES FROM ('2010-05-01') TO ('2010-06-01')  
TABLESPACE fasttablespace;
```

Há também a possibilidade de criar uma nova tabela fora da partição e fazer a conexão como uma partição da tabela particionada posteriormente. Isso permite os dados ser carregados e checados antes de ser armazenados na tabela particionada:

```
CREATE TABLE medicao_2010abr(  
LIKE medicao INCLUDING DEFAULTS INCLUDING CONSTRAINTS  
)TABLESPACE fasttablespace;
```

```
ALTER TABLE medicao_2010abr ADD CONSTRAINTS 2010abr  
CHECK ( data >= DATE '2010-04-01' AND data < DATE '2010-05-01');  
ALTER TABLE medicao ATTACH PARTITION medicao_2010abr  
FROM VALUES FROM ('2010-04-01') TO ('2010-05-01');
```

Antes de executar o comando ATTACH PARTITION é recomendado criar uma constraint CHECK na tabela particionada compatível com a partição. Dessa forma o sistema pode pular uma etapa de validar e escanear a constraint da partição, sem a constraint CHECK a tabela seria escaneada antes de ser anexada a tabela particionada.

5.11.6 Limitações

Tabelas particionadas têm certas limitações, dentre elas as principais estão listadas abaixo:

- 1: Todas os índices para partições têm de ser criados individualmente, não há maneira de criá-los automaticamente. Isso também significa que não há como criar chaves primárias, constraints UNIQUE ou constraints de exclusão que englobam todas as partições, só é possível adicionar constraints a cada partição individualmente.
- 2: Como chaves primárias não são permitidas em tabelas particionadas chaves estrangeiras referenciando tabelas particionadas também não são permitidas, nem tanto chaves estrangeira que referenciam outras tabelas dentro de tabelas particionadas.
- 3: Usar a palavra-chave ON CONFLICT para tentar forçar a união de diversas partições para adicionar constraints unicamente a todo o sistema hierárquico, esse comportamento gerará erros.
- 4: Um comando UPDATE que tenta mover uma linha para outra partição causa erros, porque o novo valor da linha falha em satisfazer a constraint da partição de origem.
- 5: Row triggers ou gatilhos de linha, se necessários devem ser adicionados às partições, não a tabela particionada.
- 6: É proibido o uso de tabelas permanentes e temporárias simulâneamente, deve se utilizar somente tabelas permanentes OU temporárias em todo o sistema hierárquico.

5.11.7 Implementação utilizando herança

Enquanto a implementação declarativa é mais intuitiva para a maioria dos casos, as vezes a implementação utilizando herança é mais vantajosa, essa abordagem têm as seguintes vantagens sobre a implementação declarativa:

- 1: Permite que a partição tenha mais objetos além da tabela particionada.
- 2: Permite herança múltipla.
- 3: Permite outras formas de particionamento além de lista e intervalo.

4: Exige menos privilégios para alteração do que a implementação declarativa. Enquanto na implementação declarativa requer o privilégio ACCESS EXCLUSIVE a para modificações a implementação utilizando herança somente os privilégios SHARE UPDATE e EXCLUSIVE são suficientes.

5.11.7.1 Tutorial de implementação utilizando herança

Implementação da mesma tabela particionada feita no tópico 5.11.4 utilizando herança.

1: Criar a tabela particionada a qual todas as outras tabelas herdarão, não defina nenhuma constraint check a não ser que queira que ela seja aplicada a todas as partições. Para a implementação desta tabela não será usada nenhuma constraint nas colunas da tabela principal.

```
CREATE TABLE medicaao (  
    city_id      int NOT NULL,  
    data        date NOT NULL,  
    temperatura int  
);
```

2: Crie as partições que herdam a tabela principal. Normalmente as partições não adicionam colunas diferentes da tabela principal. Essas tabelas serão declaradas como tabelas comuns assim como na implementação declarativa:

```
CREATE TABLE medicaao_2010fev () INHERITS (medicaao);  
CREATE TABLE medicaao_2010abr () INHERITS (medicaao);  
CREATE TABLE medicaao_2010mar () INHERITS (medicaao);
```

3: Adicione constraints não conflitantes nas partições para definir os valores chaves em cada partição.

Ex: CHECK (x = 1)

CHECK (ID_produto >= 100 AND ID_produto < 200)

Tenha certeza que as constraints são CLARAS e NÃO CONFLITEM, alguns exemplos de constraints feitas de maneira errada são:

CHECK (ID_produto BETWEEN 100 AND 200)

CHECK (ID_produto BETWEEN 200 AND 300) -ERRADO

A forma apresentada acima é errada devido ao valor chave 200 não pertencer a nenhuma das duas partições

As constraints da tabela que estamos construindo nesse exemplo são as seguintes:

```
CREATE TABLE medicao_2010fev (
CHECK (data >= DATE '2010-02-01' AND data < DATE '2010-03-01')
) INHERITS (medicao);
```

```
CREATE TABLE medicao_2010mar (
CHECK (data >= DATE '2010-03-01' AND data < DATE '2010-04-01')
) INHERITS (medicao);
```

```
CREATE TABLE medicao_2010abr (
CHECK (data >= DATE '2010-04-01' AND data < DATE '2010-05-01')
) INHERITS (medicao);
```

4: Para cada partição deve-se criar um índice na coluna chave, assim como outros índices que possam ser necessários

```
CREATE INDEX medicao_2010fev_data ON medicao_2010fev(data);
CREATE INDEX medicao_2010mar_data ON medicao_2010mar(data);
CREATE INDEX medicao_2010abr_data ON medicao_2010abr(data);
```

5: Queremos que nossa tabela redirecione os registros para a partição adequada quando realizarmos o comando INSERT INTO medicao VALUES..., isso pode ser feito usando funções de gatilho:

```
CREATE OR REPLACE FUNCTION medicao_insert_trigger()
RETURN TRIGGER AS $$
BEGIN
    INSERT INTO medicao_2010fev VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Depois de criar a função, podemos criar um gatilho que chama a função:

```
CREATE TRIGGER insert_medicao_trigger
BEFORE INSERT ON medicao
FOR EACH ROW EXECUTE PROCEDURE medicao_insert_trigger();
```

Devemos redefinir a função todo mês para que ela sempre aponte para a partição correta, o gatilho em si não precisa ser atualizado.

Caso queira que o sistema resolva automaticamente onde colocar os registros deve adicionar uma função um pouco mais complicada:

```
CREATE OR REPLACE FUNCTION medicao_insert_trigger()
RETURN TRIGGER AS $$
BEGIN
    IF ( NEW.data >= DATE '2010-02-01' AND data < DATE '2010-03-01') THEN
        INSERT INTO medicao_2010fev VALUES (NEW.*);
    ELSEIF ( NEW.data >= DATE '2010-03-01' AND data < DATE '2010-04-01')
    THEN
        INSERT INTO medicao_2010mar VALUES (NEW.*);
    ...
    ELSEIF ( NEW.data >= DATE '2010-12-01' AND data < DATE '2011-01-01')
    THEN
        INSERT INTO medicao_2010dez VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Data fora do intervalo. Corrija medicao_insert_trigger()';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Note que a função de gatilho deve ser compatível com as constraints CHECK das partições.

É ideal que se chequem primeiro as partições mais novas, para isso a função deve ser criada na ordem contrária a do exemplo acima.

Uma forma diferente de redirecionar os registros adicionados a sua respectiva partição é por meio de regras:

```
CREATE RULE medicao_insert_fev_2010 AS
ON INSERT TO medicao WHERE
    (data >= DATE '2010-02-01' AND data < DATE '2010-03-01' )
DO INSTEAD
```

```

INSERT INTO medicao_2010fev VALUES (NEW.*);

...

CREATE RULE medicao_insert_dez2010 AS
ON INSERT TO medicao WHERE
    (data >= DATE '2010-12-01' AND data < DATE '2011-01-01' )
DO INSTEAD
    INSERT INTO medicao_2010dez VALUES (NEW.*);

```

Uma regra tem muito mais *overhead* que um gatilho e não permite o uso do comando COPY. Portanto para a maioria das ocasiões é recomendada a utilização de funções de gatilhos.

6: Assim como na implementação declarativa deve-se certificar-se que o parâmetro `constraint_exclusion` está ATIVADO no arquivo `postgresql.conf`.

5.11.7.2 Manutenção de partições

Para simplesmente remover dados antigos rapidamente, delete normalmente a partição desejada:

```
DROP TABLE medicao_2010fev;
```

Para remover a partição da tabela particionada e manter o acesso a ela deve-se fazer:

```
ALTER TABLE medicao_2010fev NO INHERIT medicao;
```

Para adicionar uma nova partição para receber novos registros deve-se fazer:

```

CREATE TABLE medicao_2011jan (
    CHECK ( data >= DATE '2011-01-01' AND data < DATE '2011-02-01' )
) INHERITS (medicao);

```

Podemos também criar uma nova tabela fora da estrutura e fazer dela uma partição depois que as colunas adicionada, checadas e transformadas:

```

CREATE TABLE medicao_2011jan
(LIKE medicao INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE medicao_2011jan ADD CONSTRAINT 2011jan
CHECK ( data >= DATE '2011-01-01' AND data < DATE '2011-02-01' );
ALTER TABLE medicao_2011jan INHERIT medicao;

```

5.11.7.3 Limitações

Assim como todas as formas de particionamento o particionamento por meio da herança têm também suas limitações, é necessário levá-las em consideração ao optar ou não pelo

particionamento de uma tabela. As principais limitações do particionamento com a utilização da herança são:

1: Não há forma de verificar se todas as constraints CHECK não conflitam, a opção mais segura é criar um código que gera partições e cria ou modifica objetos relacionados que criá-los manualmente.

2: Os esquemas mostrados aqui assumem que as colunas-chave nunca mudam, ou pelo menos não mudam a ponto de ser necessário mover para outra partição. Um UPDATE que tenta fazer isso irá resultar em erro devido as constraints CHECK. Isso pode ser sobreposto por funções de gatilhos, porém isso torna o manejo dos dados muito mais complicado.

3: Se você pretende usar os comandos manuais VACUUM ou ANALYZE, não esqueça de executar o comando em cada uma das partições individualmente. A seguinte sintaxe só processaria a tabela principal:

ANALYZE medicao; -ERRADO

4: Comandos INSERT com a palavra-chave ON CONFLICT não funcionam como o esperado, devido a palavra chave ON CONFLICT só levar em consideração violações únicas na tabela direcionada, não em suas herdeiras.

5: Funções gatilho ou regras são necessárias para direcionar os dados para a partição desejada. Os gatilhos talvez sejam complicados de escrever e serão bem mais lentos que o direcionamento realizado pelo particionamento declarativo.

5.11.8 Particionamento e *constraint exclusion*

O parâmetro `constraint_exclusion` desempenha o papel de dizer ao sistema que as constraints CHECK devem ser levadas em conta durante uma consulta para que, quando provada a não necessidade de busca por uma tabela ela seja dispensada da busca.

Com isso podemos aproveitar por completo os benefícios do particionamento logo que, com as constraints CHECK que adicionamos as partições, as buscas tenham uma performance aumentada.

O padrão para o parâmetro `constraint_exclusion` não é ativado nem desativado é uma configuração intermediária chamada "*partition*" na qual o sistema tenta adivinhar quais requisições são de tabelas particionadas e quais não, assim tentando ativar essa função somente quando necessário.

A otimização por meio do parâmetro `constraint_exclusion` tem limitações, as principais são:

1: *Constraint exclusion* somente funciona quando a palavra-chave WHERE contém a coluna a qual a(s) constraints estão aplicadas. Por exemplo, uma comparando contra uma função não-imutável como `CURRENT_TIMESTAMP` não pode ser otimizada porque o sistema não saberá qual o seu valor até o tempo de execução, e portanto não poderá excluir tabelas antes da requisição.

2: É interessante que se mantenha as tabelas particionadas o mais simples quanto o possível no que se refere às constraints de partição, o uso de comparações simples para particionamento por lista, e intervalos simples para o particionamento por intervalos. Isso certifica que a *constraint_exclusion* funcionará pois ela só permite operadores indexáveis pela B-tree.

3: Todas as constraints em todas as partições da tabela particionada são levados em conta no planejamento de requisição e conforme a quantidade de partições da tabelas isso pode resultar em um *overhead* de tempo, essa limitação se faz relevante acima de centenas de partições.

5.11.9 Boas práticas do particionamento declarativo

As escolhas de como particionar uma tabela podem ser determinante no resultado final de performance das consultas.

Uma das escolhas mais importantes a se levar em conta é a escolha das colunas-chave da coluna particionada, normalmente é uma boa prática é escolher a(s) coluna(s) que mais irão aparecer em palavras-chave WHERE.

Sub-particionamento pode ser útil porém se usado demasiadamente pode levar a um *overhead* de tempo nas requisições devido ao aumento no número de tabelas.

Também é importante considerar o *overhead* no planejamento de requisições quando são utilizadas muitas partições, principalmente para os comandos UPDATE e DELETE.

O uso de grandes quantidades de partições podem ser bem utilizados no contexto de *data warehouses* e também no modelo OLTP.

5.12 Dados estrangeiros

O postgresql também implementa o uso de dados estrangeiros, que permitem o acesso do sistema postgresql a dados armazenados fora de suas dependências, o funcionamento completo será explicado posteriormente.

Há varias implementações diferentes que incluem o uso de dados estrangeiros como as bibliotecas contrib e outros produtos de terceiros.

O acesso a dados estrangeiros se dá basicamente por um “*foreign data wrapper*” que é o meio permite ao sistema o acesso aos dados. Todos os dados de configuração do “*foreign data wrapper*” são fornecidos por outra entidade chamada mapeamento de usuário.

5.13 Rastreamento de dependências

Quando se criam estruturas de bancos de dados relacionais complexas, é inevitável a dependência de alguns objetos em outros, isso gera um problema quando é necessário

excluir objetos, já que a deleção de um objeto pode fazer com que vários objetos que dependem dele funcionem de maneira inesperada ou resultem em erros.

Para assegurar a integridade dos dados nos bancos de dados o sistema do postgresql não permite a deleção de dados que têm outros objetos dependentes dentro do sistema.

Para superar essa limitação existem as palavras-chave CASCADE e RESTRICT:

A palavra-chave CASCADE deleta também todos os dados dependentes daqueles que estão sendo excluídos.

A palavra-chave RESTRICT deleta somente os objetos especificados no comando DELETE, sem distinção de se isso afetará a integridade dos dados e do funcionamento do banco de dados.

O uso das palavras-chave CASCADE e RESTRICT são obrigatórias pelo padrão SQL, porém a maioria das implementações não torna obrigatório seu uso.

Capítulo 6. Comandos DML

Os comandos anteriormente citados (DDL) são utilizados para criar tabelas e outras estruturas para armazenar dados. Os comandos a seguir (DML) são utilizados para inserir, atualizar e deletar dados dentro de tabelas.

6.1 Inserindo dados

Quando uma tabela é criada, não contém dados. Dados por padrão são inseridos uma linha de cada vez, há maneiras de inserir mais de uma linha de uma vez, porém não há como inserir menos de uma linha de uma vez.

Para criar uma nova linha use o comando INSERT. Este comando requer o nome da tabela e os valores que se deseja adicionar.

Considere a seguinte tabela:

```
CREATE TABLE produtos (  
    id_produto          integer,  
    nome                text,  
    preco               numeric  
);
```

Uma forma de inserir uma linha nessa tabela seria:

```
INSERT INTO produtos VALUES (1,'Queijo', 9.95);
```

Os valores dos dados são listados na ordem em que eles aparecem na tabela, separados por vírgulas. Normalmente os valores são constantes, mas expressões escalares também são permitidas.

A sintaxe acima tem o ônus de exigir que o usuário lembre da ordem das colunas para inserir os dados. Por exemplo, os dois comandos abaixo inserem a mesma linha que a sintaxe acima:

```
INSERT INTO produtos (id_produto, nome, preco) VALUES (1,'Queijo',9.95);
```

```
INSERT INTO produtos (nome, preco, id_produto) VALUES ('Queijo',9.95,1);
```

Muitos usuários consideram a especificação de nomes das tabelas como uma boa prática.

Se o usuário deseja adicionar valores a todas as colunas ainda pode adicionar somente as colunas desejadas e o restante das colunas será preenchido com os valores default:

Também é possível requisitar manualmente o preenchimento das colunas em branco com os valores DEFAULT:

```
INSERT INTO produtos (id_produto, nome, preco) VALUES (1, 'Queijo', DEFAULT);
```

INSERT INTO produtos DEFAULT VALUES;

Assim como também é possível inserir várias linhas ao mesmo tempo:

```
INSERT INTO produtos (id_produto, nome, preco) VALUES  
(1, 'Queijo', 9.95),  
(2, 'Maçã', 2.88),  
(3, 'Pera', 3.55);
```

O resultado de uma consulta também pode ser inserido diretamente em uma tabela:

```
INSERT INTO produtos (id_produto, nome, preco)  
SELECT id_produto, nome, preco FROM novos_produtos  
WHERE release_date = 'today';
```

Ao inserir grandes quantidades de linhas nas tabelas considere utilizar o comando COPY. Não é tão flexível quanto o comando INSERT mas é mais eficiente e tem melhor performance

6.2 Atualizando dados

A mudança de dados que já foram adicionados na tabela se chama atualização. Os usuários podem atualizar uma linha ou um conjunto de linhas, uma coluna ou um conjunto de colunas. As colunas que forem selecionadas não são afetadas.

Para atualizar linhas existentes use o comando UPDATE. Este comando exige as seguintes informações:

- 1: O nome da tabela e da coluna a serem atualizados.
- 2: O novo valor da coluna
- 3: Quais linhas a serem atualizadas

Relembre que o SQL não disponibiliza nenhuma forma de identificação única de linhas. Portanto deve-se especificar condições para as linhas a serem atualizadas, se há *constraints primary key* na tabela é possível especificá-la para identificar linhas individualmente.

Por exemplo, um comando para atualizar todos os produtos com o preço 5 seria:

```
UPDATE produtos SET preco = 10 WHERE preco = 5;
```

Isso pode causar atualização em uma, nenhuma ou mais de uma linhas. Tentar atualizar com uma condição a qual nenhuma linha atende não resulta em erros.

Vamos olhar para o comando em detalhes. Primeiramente a palavra chave UPDATE seguida do nome da tabela, depois a palavra-chave SET, seguida do nome da coluna, um

sinal de igual (que neste caso significa atribuição) e o novo valor da coluna. O novo valor da coluna pode ser qualquer expressão escalar como, por exemplo:

```
UPDATE produtos SET preco = preco * 1.10;
```

O código acima aumenta o preço de todos produtos em 10%.

Se a cláusula WHERE é omitida, todas as linhas serão atualizadas, se presente somente as que atendem aos requisitos apresentados nela.

Também podem ser atualizadas mais de uma coluna de uma vez como o exemplo abaixo:

```
UPDATE tabela1 SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.2 Excluindo dados

Como já foram especificados como adicionar e atualizar dados em tabelas o próximo tópico lógico explica como excluir linhas em tabelas. Assim como só é possível adicionar linhas inteiras, também só é possível excluir linhas inteiras. Assim como mencionado no tutorial anterior o sistema não dispõe de qualquer identificação individual para linhas. Portanto é necessária a especificação de condições na palavra-chave WHERE.

A sintaxe usada no comando DELETE para excluir linhas é muito parecida com a sintaxe utilizada para atualizar linhas utilizando o comando UPDATE. Por exemplo, para deletar todas as linhas da tabela produtos com o preço 10 é usado o seguinte comando:

```
DELETE FROM produtos WHERE preco = 10;
```

Se, ao invés do comando acima você por engano executar:

```
DELETE FROM produtos; -ERRADO
```

Excluiria todas as linhas da tabela. E este comando seria, na maioria das vezes, irreversível.

6.3 Obtendo dados de linhas modificadas

Às vezes é necessário o retorno dos dados de uma tabela que estão sendo modificados. Para esses casos existe a palavra chave RETURNING.

O principal motivo da utilização desta palavra-chave é dispensar uma possível consulta posterior a tabela para revisar os dados modificados e é ainda mais efetiva se os dados modificados possam ficar indisponíveis após a realização da alteração. RETURNING funciona para todos os comandos DML.

As seleções de retorno usando a palavra-chave RETURNING são as mesmas do comando SELECT.

Ao utilizar a palavra-chave RETURNING juntamente com o comando INSERT, os dados retornados pela palavra-chave são as linhas inseridas.

**Ex:INSERT INTO tabela1 (nome, idade) VALUES
('João', 15) RETURNING *;**

RETORNO

Nome	Idade
João	15

Ao utilizar a palavra-chave RETURNING juntamente com o comando UPDATE, os dados retornados são as linhas modificadas.

**Ex:UPDATE tabela1 SET nome = 'Pedro' WHERE nome = 'João'
RETURNING *;**

RETORNO

Nome	Idade
Pedro	15

Ao utilizar a palavra-chave RETURNING juntamente com o comando DELETE, os dados retornados são as linhas deletadas.

**Ex:DELETE FROM tabela1 WHERE nome = 'Pedro'
RETURNING *;**

RETORNO

Nome	Idade
Pedro	15

Capítulo 7. Comando DQL

O processo de pesquisar dados de um banco de dados é chamado consulta. Em SQL o comando SELECT é usado para realizar consultas.

7.1 Conceitos gerais

A sintaxe usada neste comando é a seguinte:

```
SELECT lista_de_selecao FROM expressao_de_tabela ;
```

As seções adiante descrevem os detalhes da lista de seleção, expressões de tabelas e expressões de ordenação. A palavra-chave WITH será explicada posteriormente pois é um recurso avançado.

Uma consulta simples pode ser realizada da seguinte forma:

```
SELECT * FROM tabela1;
```

Assumindo que há uma tabela com o nome tabela1, (O resultado pode variar depende do client usado para acessar o banco, por exemplo, o psql mostrará uma representação visual da consulta usando ASCII). A lista de seleção * significa todas as colunas que a tabela possui.

Uma lista de seleção também pode ser um grupo de colunas como, por exemplo, se a tabela1 tem colunas de nome a, b e c a seguinte consulta poderia ser feita:

```
SELECT a, b + c FROM tabela1;
```

(Assumindo que b e c são de algum tipo de dado numérico.

FROM tabela1 é um tipo simples de expressão de tabela, ela pesquisa apenas uma tabela. Em geral, expressões de tabela podem ser construções complexas de tabelas como junções e sub-consultas. Mas também é possível omitir a expressão de tabela e usar o comando SELECT como uma calculadora:

```
SELECT 3 * 4;
```

Isso se torna mais útil se as expressões na lista de seleção retornam valores variáveis. Por exemplo, uma função poderia ser chamada desta forma:

```
SELECT random();
```

7.2 Expressões de tabela

As expressões de tabela contém a(s) tabelas a ser(em) consultada(s), elas são precedidas pela palavra-chave FROM e são opcionalmente seguidas por WHERE, GROUP BY ou HAVING. Uma expressão de tabela trivial somente referencia uma tabela

no disco, mas expressões mais complexas podem ser utilizadas para relacionar e combinar tabelas de várias formas.

As palavras-chave opcionais WHERE, GROUP BY e HAVING têm a funcionalidade de realizar uma sequência transformações na tabela informada na palavra-chave FROM. Essas transformações originam uma tabela virtual originada a partir da tabela apresentada porém com as alterações feitas por essas palavras-chave que é retornada como resultado da consulta.

7.3 A palavra-chave FROM

A palavra chave FROM é seguida por uma expressão de tabela, que por sua vez é feita a partir de uma ou mais referencias de tabelas separadas por vírgula.

FROM referencia_de_tabela (, nometabela2, nometabela3 ...)

Uma referência de tabela pode ser um nome de tabela (possivelmente nome qualificado) ou uma tabela derivada, como uma sub-consulta, ou uma junção, ou combinações mais complexas destes. Referencias de tabelas sem nenhuma junção especificada somente separadas por vírgula são unidas por meio da junção CROSS-JOINT para formar a tabela virtual de retorno.

O resultado da lista de referencias de tabelas é uma tabela virtual intermediária que pode então sofrer transformações feitas pelas palavras-chave WHERE, GROUP BY e HAVING.

Se a tabela presente na referência de tabela é uma tabela herdada, a consulta mostrará também os dados das tabelas herdeiras porém somente colunas da tabela herdada, colunas adicionadas somente as tabelas-filhas não serão mostradas. Para evitar isso deve ser usada a palavra-chave ONLY.

Também pode-se utilizar o asterisco (*) após o nome da tabela para especificar a consulta também em tabelas herdeiras, este tipo de sintaxe é recomendado somente quando se deseja ter compatibilidade com versões anteriores já que atualmente este é o comportamento padrão das consultas quando a palavra-chave ONLY é omitida.

7.4 Junções entre tabelas

Uma junção de tabelas é feita a partir de duas outras (reais ou derivadas) de acordo com o tipo de junção, sendo eles: INNER, OUTER e CROSS-JOINS. A sintaxe geral para as junções de tabelas é:

tabela1 tipo_de_junção tabela2 [condição de junção]

Junções de todos os tipos podem ser concatenadas ou aninhadas: tabela1 e tabela2 podem ser junções. Parênteses podem ser usadas para controlar a ordem das junções. Caso não haja parênteses a ordem das junções é da esquerda para a direita.

Há diversos tipos de junções possíveis entre tabelas sendo elas:

7.4.1 CROSS-JOIN

tabela1 CROSS JOIN tabela2;

Para cada combinação possível de linhas entre tabela1 e tabela2, a junção vai conter uma linha consistindo de todas as colunas na tabela1 seguida de todas as colunas na tabela2. Se as tabelas contêm quantidades X e Y de linhas respectivamente, a junção terá X * Y linhas.

FROM tabela1 CROSS JOIN tabela2;

é equivalente a:

FROM tabela1 INNER JOIN tabela2 ON TRUE;

e também a:

FROM tabela1, tabela2;

7.4.2 Junções qualificadas

T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2

ON boolean_expression;

T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING

(join column list);

T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2;

As palavras INNER e OUTER são opcionais em todas as formas. INNER é o padrão. LEFT, RIGHT, e FULL só se aplicam a junções OUTER.

A condição de junção é especificada junto com a palavra-chave ON ou USING, ou implicitamente na palavra-chave NATURAL. A condição de junção determina quais linhas das duas tabelas são verificadas se são "compatíveis" como explicado abaixo.

Os tipos de junções qualificadas são:

INNER JOIN: Para cada linha da tabela1, a junção tem uma linha caso haja uma linha na tabela2 que satisfaz a condição de junção com a linha da tabela1.

LEFT OUTER JOIN: Primeiro uma INNER JOIN é feita. Então, para cada linha da tabela1 que não satisfaz a condição de junção com nenhuma linha na tabela2, uma linha é adicionada com valores NULL nas colunas da tabela2. Portanto esta junção sempre terá ao menos uma linha para cada linha na tabela1.

RIGHT OUTER JOIN: Primeiro é feita uma INNER JOIN. Então, para cada linha na tabela2 que não satisfaz a condição de junção com nenhuma linha na tabela1, uma linha é adicionada com valores NULL nas colunas da tabela1. Portanto esta junção sempre terá ao menos uma linha para cada linha na tabela2.

FULL OUTER JOIN: Esta junção é o oposto de uma INNER JOIN, seleciona somente as linhas da tabela1 que não possuem correspondentes que satisfazem a condição de junção com a tabela2, adicionando valores NULL nas colunas da tabela2 e vice-versa.

A palavra-chave ON é a forma de condição geralmente utilizada, ela recebe uma expressão booleana do mesmo tipo utilizado na palavra-chave WHERE porém utilizando um par de linhas compatíveis resulta em um valor TRUE na expressão.

A palavra-chave USING é um atalho utilizado em uma situação específica quando as duas colunas a serem comparadas têm o mesmo nome. Esta palavra-chave pega os nomes de colunas de nomes iguais a serem comparadas nas duas tabelas separados por vírgula e forma uma condição de união que inclui uma comparação de igualdade.

Fazendo a junção entre as tabelas T1 e T2 com a palavra chave USING (a, b) seria equivalente à seguinte condição de junção:

ON T1.a = T2.a AND T1.b = T2.b;

Além disso o uso de USING dispensa os resultados redundantes. Enquanto ON exhibe tanto os resultados da tabela1 quanto os da tabela2. USING exhibe os valores compatíveis somente uma vez.

Finalmente o uso de NATURAL é um atalho para a palavra-chave USING, que faz, automaticamente, o equivalente a USING formando por si mesma uma condição de junção entre as colunas de mesmo nome presentes nas duas tabelas. Se não há colunas de mesmo nome nas tabelas a junção ocorre como JOIN ... ON TRUE, produzindo uma CROSS-JOIN.

USING é segura contra mudanças de esquema nas tabelas já que as colunas a serem comparadas estão escritas. Já a palavra-chave NATURAL pode produzir resultados inesperados com a adição de uma nova coluna de nome compatível.

7.5 Apelidos de tabelas e colunas

Um nome temporário pode ser dado a tabelas com nomes longos e este nome temporário pode ser usado para se referir à tabela ou coluna durante toda a consulta. Estes nomes são chamados de aliases e podem ser criados seguindo a seguinte sintaxe:

FROM nometabela AS alias

ou

FROM nometabela alias

Uma vez criado o alias o nome original da tabela não pode mais ser usado em lugar algum da consulta. Os aliases normalmente são uma opção meramente mais conveniente, porém em alguns casos são obrigatórios como:

1: Junção de uma tabela com ela mesma

Ex: **SELECT * FROM pessoas AS mae JOIN pessoas AS filha**
ON mae.id = filha.mae_id;

2: Consultas utilizando sub-consultas

Ex: **SELECT * FROM (my_table1 AS a CROSS JOIN my_table2 AS b) ...**

Há também como criar aliases para colunas, seguindo a seguinte sintaxe:

FROM nometabela alias (aliascoluna1 , aliascoluna2 ...);

Se o número de aliases é menor que o de colunas o restante das colunas não recebe aliases.

7.6 Sub-consultas

Sub-consultas especificando uma tabela derivada devem estar entre parênteses e devem receber aliases. Por exemplo:

FROM (SELECT * FROM nometabela) AS alias

Que seria equivalente a:

FROM nometabela AS alias

Casos mais interessantes que não podem ser reduzidos a uma junção plana surgem quando a sub-consulta envolve agrupamento ou agregação de tabelas.

Uma sub-consulta pode também receber uma lista de valores:

FROM (VALUES ('Joaquim', 'Silva'), ('Pedro', 'Gonçalves'))

AS nomes (primeiro, sobrenome);

7.7 Funções de tabela

Funções de tabela produzem um grupo de linhas, feitas de tipos de dados base ou tipos de dados compostos (linhas de tabelas). Elas são usadas como tabelas na cláusula FROM. Colunas retornadas por funções de tabela podem ser utilizadas também nos comandos SELECT, JOIN e WHERE.

Funções de tabelas também podem ser combinadas utilizando a sintaxe ROWS FROM com os resultados retornando em colunas paralelas, o resultado neste caso será o mesmo do resultado na maior função, o menor resultado será preenchido com valores NULL.

function_call [WITH ORDINALITY opcional] [[AS] tabela_alias [(coluna_alias
[, ...])]
ROWS FROM(function_call [, ...]) [WITH ORDINALITY opcional]
[[AS] tabela_alias [(coluna_alias [, ...])]]

Se o comando WITH ORDINALITY é utilizada, uma coluna adicional do tipo bigint vai ser adicionada as colunas do resultado. Essa coluna enumera do resultado começando de 1. Esta coluna por padrão se chama ordinality porém pode receber aliases.

A função de tabela especial UNNEST pode ser chamada com qualquer número de parâmetros array, e ela retorna o número de colunas, como se UNNEST fosse chamado em cada parâmetro separadamente e depois combinado usando a construção ROWS FROM.

UNNEST(array_expression [, ...]) [WITH ORDINALITY]

[[AS] tabela_alias [(coluna_alias [, ...])]]

Se os aliases não são fornecidos, então para uma função retornando um tipo de dado base, o nome da coluna é sempre o nome da função. Para uma função retornando um tipo composto, as colunas do resultado recebem o nome dos atributos individuais dos tipos.

Ex: CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int)

RETURNS SETOF foo AS \$\$

SELECT * FROM foo WHERE fooid = \$1;

\$\$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo

WHERE foosubid IN (

SELECT foosubid

FROM getfoo(foo.fooid) z

WHERE z.fooid = foo.fooid

);

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;

Em alguns casos o usuário pode querer que a função retorne resultados de tipos de dados diferentes dependendo de como eles são chamados, para isso a função deve retornar o valor record, isso significa que, quando utilizada em uma consulta, a estrutura do resultado desejado deve ser especificada. Assim como o exemplo abaixo:

function_call [AS] alias (column_definition [, ...])

function_call AS [alias] (column_definition [, ...])

ROWS FROM(... function_call AS (column_definition [, ...])

[, ...])

Quando a primeira sintaxe é utilizada (sem a presença da palavra-chave `ROWS FROM` a lista `column_definition` substitui a coluna `alias` que pode ser adicionada a palavra-chave `FROM`, os nomes nas definições de colunas funcionam como aliases.

Quando a segunda sintaxe é utilizada a lista `column_definition` pode ser adicionada a cada membro da função separadamente, ou, se há a presença de somente um membro e sem a palavra-chave `WITH ORDINALITY` a lista `column_definition` pode substituir a lista de aliases após `ROWS FROM()`.

Ex: **SELECT ***
FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';

7.8 Sub-consultas laterais

Sub-consultas aparecendo em `FROM` podem ser precedidas pela palavra-chave `LATERAL`, ela permite que uma sub-consulta pode referenciar colunas da sub-consulta à sua esquerda.

Funções de tabela após a palavra-chave `FROM` também podem ser precedidas pela palavra-chave `LATERAL`, mas para funções a mesma é opcional, já que os argumentos da função podem conter referência a colunas à esquerda da palavra-chave `FROM` de qualquer forma.

A palavra-chave `LATERAL` pode estar também presente dentro de junções, além da cláusula `FROM`, neste caso ela é utilizada para permitir que a tabela direita da junção possa acessar colunas do lado esquerdo da junção.

Segue um exemplo do uso da palavra-chave lateral:

```
FROM SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id =  
foo.bar_id) ss;
```

Este não é um exemplo útil já que ele tem o mesmo resultado que esta sintaxe mais simples abaixo:

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

7.9 A cláusula WHERE

A sintaxe da cláusula `WHERE` é a seguinte:

```
WHERE condição_de_pesquisa
```

A condição de pesquisa é qualquer expressão que retorne um valor booleano (TRUE ou FALSE).

Uma vez que o processamento da palavra-chave FROM é feito, uma tabela virtual com os resultados é criada e cada linha desta tabela é testada pela expressão presente na cláusula WHERE, os que resultarem em NULL ou FALSE são retirados do retorno da consulta.

A condição de junção de uma INNER JOIN pode ser escrita tanto na cláusula WHERE ou na cláusula JOIN. Ex:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

e:

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

ou também:

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Agora alguns exemplos da sintaxe de cláusulas WHERE:

```
SELECT ... FROM fdt WHERE c1 > 5
```

```
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 =  
    fdt.c1 + 10)
```

```
SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 =  
    fdt.c1 + 10) AND 100
```

```
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 >  
    fdt.c1)
```

Repare como a coluna c1 é referenciada por seu nome qualificado nas sub-consultas, isso só é necessário caso haja alguma coluna de mesmo nome na sub-consulta, porém esta abordagem deixa os comandos mais legíveis.

7.10 As cláusulas GROUP BY e HAVING

Logo após a cláusula WHERE, a tabela virtual de retorno da consulta pode ser sujeita a agrupamento, usando a cláusula GROUP BY, e eliminação de grupos de linhas utilizando HAVING.

Ex:

```
SELECT lista_de_selecao
```

FROM ...

[WHERE ... opcional]

GROUP BY referencia_de_agrupamento

[, referencia_de_agrupamento]...

A cláusula GROUP BY é usada para agrupar as linhas na tabela que têm os mesmo valores em todas as colunas listadas. A ordem na qual as colunas são listadas não importa. O efeito é combinar cada grupo de linhas que tem valores em comum em uma linha de grupo que representa todas as linhas do grupo.

Isso é feito para eliminar redundâncias na resposta da consulta e/ou computar expressões de agregação que se apliquem a esses grupos.

=# SELECT * FROM test1;

x	y
a	3
c	2
b	5
a	1

=# SELECT x FROM test1 GROUP BY x;

x
a
b
c

Na segunda consulta não poderia ser escrita da seguinte forma:

SELECT * FROM test1 GROUP BY x; – ERRADO

Porque não há nenhum valor para a coluna y que pudesse ser associado a cada grupo. As linhas agrupadas podem ser referencia na lista de seleção desde que elas tenham somente um valor para cada grupo.

Em geral, se a tabela é agrupada, colunas que não são listadas na cláusula GROUP BY não podem ser referenciadas exceto em expressões de agregação. Um exemplo com expressões agregadas é:

=# SELECT x, sum(y) FROM test1 GROUP BY x;

x	sum
a	4

b	5
c	2

Neste caso a função `sum()` computa um valor para cada linha agrupada.

No exemplo a seguir uma consulta calcula o total de vendas para cada produto (ao invés das vendas totais de todos os produtos):

```
SELECT id_produto, p.nome, (sum(s.unidades) * p.preco) AS vendas
FROM produtos p LEFT JOIN vendas s USING (id_produto)
GROUP BY id_produto, p.nome, p.preco;
```

Neste exemplo, as colunas `id_produto`, `p.nome` e `p.preco` devem estar na cláusula `GROUP BY` porque eles são referenciados na lista de seleção. A coluna `s.unidades` não tem de estar na lista `GROUP BY` desde que é só utilizada na função de agregação `sum()`.

Se a coluna `id_produto` for a chave primária então será suficiente agrupar somente a mesma, porque o nome e o preço seriam dependentes do `id_produto`, portanto, não apresentariam ambiguidades sobre qual nome e valor seriam apresentados para cada grupo de `id_produto`.

A norma padrão `sql` `GROUP BY` pode somente agrupar colunas da tabela fonte, mas o `postgresql` estende essa cláusula para também permitir o agrupamento de colunas na lista de seleção, além do agrupamento por expressões.

Se uma tabela foi agrupada usando `GROUP BY`, mas somente alguns grupos são de interesse, a cláusula `HAVING` pode ser utilizada, de modo parecido com a cláusula `WHERE`, para eliminar grupos do resultado. A sintaxe é:

```
SELECT lista_de_selecao FROM ... [WHERE ...] GROUP BY ...
HAVING expressão
```

As expressões após `HAVING` podem se referir tanto a expressões agrupadas e a expressões desagrupadas (que necessariamente tem que envolver uma função de agregação)

```
=# SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
```

x	sum
a	4
b	5

```
=# SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
```

x	sum
---	-----

a	4
b	5

Um exemplo prático:

```
Ex:  SELECT id_produto, p.nome, (sum(s.unidades) * (p.preco - p.custo)) AS
      lucro
      FROM produtos p LEFT JOIN vendas s USING (id_produto)
      WHERE s.data > CURRENT_DATE - INTERVAL '4 weeks'
      GROUP BY id_produto, p.nome, p.preco, p.custo
      HAVING sum(p.preco * s.unidades) > 5000;
```

No exemplo acima, a cláusula `WHERE` está selecionando linhas por uma coluna que não está presente na cláusula de agrupamento, enquanto a cláusula `HAVING` restringe o resultado a grupos em que as vendas são maiores que 5000.

Se uma consulta tem funções de agregação mas não contém a cláusula `GROUP BY` o agrupamento ainda acontece, mas agora o resultado é só uma linha agrupada. Isso também ocorre se a consulta contém uma cláusula `HAVING` mesmo sem `GROUP BY`.

7.11 GROUPING SETS, CUBE e ROLLUP

Operações de agrupamento mais complexas podem ser possíveis usando *grouping sets*. Os dados selecionados pelas palavras-chave `FROM` e `WHERE` são agrupadas separadamente por cada *grouping set*, agregações computadas para cada grupo assim como nas cláusulas `GROUP BY` normais e então o resultado é retornado.

Como por exemplo:

```
=# SELECT * FROM items_sold;
```

Marca	Tamanho	Vendas
Foo	G	10
Foo	M	20
Bar	M	15
Bar	G	5

```
=# SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING
      SETS ((brand), (size), ());
```

Marca	Tamanho	sum
Foo		30

Bar		20
	L	15
	M	35
		50

Cada sub-lista de GROUPING SETS deve especificar 0 ou mais colunas ou expressões e é interpretada da mesma forma que se fosse diretamente na palavra-chave GROUP BY. Um *grouping set* significa que todas as linhas são agregadas em um só grupo (que é retornado mesmo se não contiver nenhuma linha).

Referências para colunas de agrupamento ou expressões são substituídas por valores nulos no resultado para *grouping sets* nos quais aquelas colunas não aparecem.

Há também uma notação encurtada para especificar dois tipos comuns de *grouping sets*:

ROLLUP (e1, e2, e3 ...)

Representa a seguinte lista de expressões e todos os prefixos da lista incluindo a lista vazia. Portanto é equivalente a:

GROUPING SETS (

(e1, e2, e3, ...),

...

(e1, e2),

(e1),

()

);

Essa sintaxe é normalmente usada para analisar dados hierárquicos, como salário total por departamento, divisão , e o total da companhia.

Já a notação CUBE

CUBE (e1, e2, e3,...)

Representa a lista dada como parâmetro e todas os subconjuntos possíveis. Portanto:

CUBE (a, b, c)

é equivalente a:

GROUPING SETS (

(a, b, c),

(a, b),

```
(a, c),  
(a ),  
( b, c),  
( b ),  
( c),  
( )  
);
```

7.12 Listas de seleção

Como mostrado anteriormente o as expressão de tabela no comando SELECT criam uma tabela virtual intermediária combinando tabelas, views, eliminando linhas, agrupando, etc. Essa tabela então é passada para o processamento da lista de seleção. A lista de seleção determina quais colunas da tabela intermediária realmente aparecerão no resultado.

7.12.1 Itens da lista de seleção

Os itens da lista de seleção são as colunas que realmente serão mostradas, a lista mais simples possível contém somente o elemento *, que resulta em todas as colunas.

A lista de seleção também pode conter nomes de colunas, porém as tabelas as quais elas pertencem devem estar listadas na palavra-chave FROM.

Quando utilizadas mais de uma tabela pode-se ocorrer de duas tabelas terem colunas de nomes idênticos, então deve-se utilizar o nome qualificado de cada coluna:

```
SELECT tabela1.a, tabela2.a FROM...
```

7.13 Rótulos de colunas

As colunas selecionadas para o resultado podem receber novos nomes após o processamento, como para o uso na cláusula ORDER BY ou para apresentar ao cliente da aplicação, como no exemplo

```
SELECT a value, b + c AS sum FROM ...
```

Repare que na primeira operação de nomeação a palavra AS não é utilizada para rotular a coluna a com o nome value, ao contrário da segunda. Isso porque a palavra-chave AS é opcional. Porém o comando acima seria inválido devido ao novo nome dado a coluna a é uma palavra-chave existente no sistema Postgresql, para evitar esse tipo de coincidência acidental deve-se utilizar aspas duplas no nome da coluna:

```
SELECT a "value", b + c AS sum FROM ...
```

Dessa forma o comando executará normalmente.

7.14 A palavra chave DISTINCT

A palavra chave DISTINCT elimina todas as linhas consideradas idênticas, as linhas idênticas são linhas com valores iguais em todas as colunas, valores NULL são considerados idênticos nessa comparação, a sintaxe do uso da palavra-chave DISTINCT é:

```
SELECT DISTINCT lista_de_selecao ...
```

Há também a palavra-chave ALL, que faz o comportamento contrário e tem a mesma sintaxe (anterior a lista de seleção). Porém é redundante devido ao comportamento normal da consulta seja esse.

7.15 Combinação de consultas

Os resultados de duas consultas pode ser combinado usando as operações de união, intersecção e diferença. A sintaxe é:

```
consulta1 UNION consulta2
```

```
consulta1 INTERSECT consulta2
```

```
consulta1 EXCEPT consulta2
```

UNION somente une o resultado da consulta2 com o resultado da consulta1 e elimina resultados repetidos, da mesma forma que a palavra-chave DISTINCT, a menos que UNION ALL seja usado. (Não há garantia da ordem do resultado)

INTERSECT retorna todas as linhas que estão presentes nas duas consultas e elimina resultados duplicados, a menos que INTERSECT ALL seja usado.

EXCEPT retorna todas as linhas que estão presentes na consulta1 que não estão presentes na consulta2 e, também, elimina resultados duplicados, a menos que EXCEPT ALL seja usado. (Também é chamado de diferença entre duas consultas).

Para calcular a UNION, INTERSECT e EXCEPT de duas consultas elas devem retornar o mesmo número de colunas e as colunas correspondentes devem ter dados compatíveis.

7.16 Ordenando linhas

Após uma consulta produzir uma uma tabela de retorno ela pode ser opcionalmente ordenada, se nenhuma ordenação é escolhida, as linhas são retornadas em uma ordem não especificada, que depende de de vários fatores como a ordem dos dados no disco.

A cláusula ORDER BY especifica a ordenação:

```
SELECT lista_de_selecao
```

```
FROM expressao_de_tabela
```

ORDER BY expressão_de_ordenação [ASC | DESC] [NULLS { FIRST | LAST}] [, expressão_de_ordenação [ASC | DESC] [NULLS { FIRST | LAST}]...]

A expressão de ordenação pode ser qualquer expressão que seja válida na lista de seleção da consulta, um exemplo é:

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

Quando mais de uma expressão é especificada os valores a direita ordenam as linhas consideradas iguais pelas expressões a direita. Cada expressão pode ser seguida pelas palavras ASC e DESC. ASC ordena os valores de forma ascendente, ou seja, do menor para o maior, e DESC faz o contrário.

As opções NULLS FIRST ou NULLS LAST são usadas para determinar se os valores nulos aparecem antes ou depois dos valores não nulos na ordenação. Por padrão, valores nulos são considerados maiores que qualquer valor não nulo, portanto, se a ordem está DESC o padrão é NULLS FIRST e, para ASC, o padrão é NULLS LAST.

ATENÇÃO!. As opções de ordenação são independentes para cada expressão de ordenação. Por exemplo:

ORDER BY x, y DESC é igual a ORDER BY x ASC, y DESC

ORDER BY x, y DESC é diferente de ORDER BY x DESC, y DESC

Uma expressão de ordenação também pode ser um rótulo de coluna ou o número de uma coluna na lista de seleção:

SELECT a + b AS sum, c FROM table1 ORDER BY sum;

SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;

O nome de uma coluna de retorno não pode ser uma expressão, deve ser um nome individual, não uma expressão, o comando a seguir resultaria em um erro:

SELECT a + b AS sum, c FROM table1 ORDER BY sum + c; --ERRADO

7.17 LIMIT e OFFSET

LIMIT e OFFSET possibilitam o retorno de somente uma porção das linhas que são geradas pela consulta:

```
SELECT lista_de_selecao  
FROM expressao_de_tabela  
[ ORDER BY ... ]  
[ LIMIT {número | ALL} ] [ OFFSET número ]
```

Se um limite é especificado não serão retornadas mais linhas que o especificado (mas podem ser menos, se o resultado da consulta for menor), LIMIT ALL é o mesmo que omitir a palavra-chave.

OFFSET omite o número especificado de linhas antes de começar a retornar o resultado da consulta.

Quando utilizando LIMIT e OFFSET ao mesmo tempo, as linhas omitidas pelo OFFSET não contam para a palavra-chave LIMIT.

É importante utilizar a cláusula ORDER BY para organizar as linhas antes de utilizar LIMIT ou OFFSET, para que não se excluam linhas de forma aleatória, já que sem a cláusula ORDER BY a ordem do resultado da consulta é indefinido.

7.18 Lista VALUES

A palavra-chave VALUES é uma forma de gerar uma tabela de constantes que pode ser usada em uma consulta sem ter que armazená-la em disco. A sintaxe é:

VALUES (expressão [, ...]) [, ...]

Cada lista de expressões entre parênteses gera uma linha na tabela. As lista devem todas ter o mesmo número de colunas, e os registros nestas listas devem ter tipos compatíveis. O tipo de dados atribuído para cada coluna do resultado é determinado usando as regras UNION.

Como no seguinte exemplo:

VALUES (1, 'one'), (2, 'two'), (3, 'three');

Vai retornar uma tabela de duas colunas e três linhas. É equivalente a:

SELECT 1 AS column1, 'one' AS column2

UNION ALL

SELECT 2, 'two'

UNION ALL

SELECT 3, 'three';

Por padrão, Postgresql atribui os nomes column1, column2, etc. as colunas de uma tabela VALUES. Os nomes destas colunas não é especificado pelo padrão SQL portanto é uma boa prática sobrescrever os nomes padrão com aliases, como o seguinte exemplo:

SELECT * FROM (VALUES (1, 'one'), (2,'two'), (3, 'three')) AS t (num, letter);

num	letter
1	one
2	two
3	three

Sintaticamente, VALUES seguido por uma lista de expressões é tratado de forma equivalente a:

SELECT select_list FROM table_expression;

e pode aparecer em qualquer lugar que uma consulta SELECT pode. Por exemplo, você pode usá-las como parte de uma união, ou combinar com uma especificação de ordenação (ORDER BY, LIMIT, e/ou OFFSET). lista VALUES são comumente usadas como fonte para um comando INSERT ou como uma sub-consulta.

Capítulo 8. Tipos de dados

Os tipos de dados são divididos entre diversas classes, sendo elas numéricos, monetários, caracteres, binários, data/hora, boolean, enumerações, geométricos, endereços de rede, bit string, pesquisa de texto, UUID, XML, JSON, Arrays, tipos compostos, *Range types*, identificadores de objeto, pg_isn, pseudo-tipos entre outros, a intenção deste capítulo é somente servir como um guia de referência para cada um dos tipos mais importantes de dados para que, cada vez que necessários possam ser rapidamente acessados.

8.1 Numéricos

NOME	TAMANHO	DESCRIÇÃO	RANGE
smallint	2 bytes	Inteiros curtos	-32768 a +42767
Integer	4 bytes	Inteiros de tamanho normal	-2147483648 a +2147483647
bigint	8 bytes	Inteiros grandes	-2147483648 to +2147483647
decimal	Variável	Decimais com precisão determinada pelo usuário	aproximadamente 131072 dígitos decimais; aproximadamente 16383 dígitos não decimais
real	4 bytes	Reais com pouca precisão	6 dígitos decimais
double precision	8 bytes	Inteiros com precisão média	15 dígitos decimais
smallserial	2 bytes	Pequeno inteiro com auto-incrementação	1 a 32767
serial	4 bytes	Inteiro com auto-incrementação	1 a 2147483647
bigserial	8 bytes	Inteiro grande com auto-incrementação	1 a 9223372036854775807

8.2 Caracteres

NOME	TAMANHO	DESCRIÇÃO
varchar(n)	Variável	String variável com limite fixo
char(n)	Variável	String de extensão fixa completada com espaços em branco
text	Variável	String variável sem limite fixo

8.3 Data e tempo

NOME	TAMANHO	DESCRIÇÃO	RANGE
timestamp(d)	8 bytes	Data e tempo sem fuso horário	4713 AC a 294276 DC
timestamp(d+f)	8 bytes	Data e tempo com fuso horário	4713 AC a 294276 DC
date	4 bytes	Data sem hora	4713 AC a 5874897 DC
time(t)	8 bytes	Hora sem data e sem fuso horário	00:00:00 a 24:00:00
time(t+f)	12 bytes	Hora sem data e sem fuso horário	00:00:00+1459 a 24:00:00-1459
interval	16 bytes	Intervalo de tempo	-178000000 anos a 178000000 anos

8.3 Booleano

NOME	TAMANHO	DESCRIÇÃO
Boolean	1 byte	Verdadeiro ou falso

8.4 Demais tipos

NOME	TAMANHO	DESCRIÇÃO	RANGE
money	8 bytes	Valores monetários	-92233720368547758.08 a +92233720368547758.07
cidr	7/9 bytes		