6) Utilizando Operadores

6.1) Introdução aos operadores

Um operador é algo que você alimenta com um ou mais valores e que devolve outro valor.

Operador é quem liga duas constantes ou variáveis. Temos operadores matemáticos, de string, de data, de atribuição, lógicos, de array, etc.

Seguem alguns exemplos.
Exemplo - Resolução do tipo em operador de exponenciação

Existe apenas um operador de exponenciação definido no catálogo, e recebe argumentos do tipo double precision. O rastreador atribui o tipo inicial integer aos os dois argumentos desta expressão de consulta:

```
=> SELECT 2 ^ 3 AS "Expressão";

 expressao
-----
   8
(1 linha)
```

Portanto, o analisador faz uma conversão de tipo nos dois operandos e a consulta fica equivalente a

```
=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "exp";
```

Exemplo - Resolução do tipo em operador de concatenação de cadeia de caracteres

Uma sintaxe estilo cadeia de caracteres é utilizada para trabalhar com tipos cadeias de caracteres, assim como para trabalhar com tipos de extensão complexa. Cadeias de caracteres de tipo não especificado se correspondem com praticamente todos os operadores candidatos.

Um exemplo com um argumento não especificado:

```
=> SELECT text 'abc' || 'def' AS "texto e desconhecido";

 texto e desconhecido
----------------------
 abcdef
(1 linha)
```

Neste caso o analisador procura pela existência de algum operador recebendo o tipo text nos dois argumentos. Uma vez que existe, assume que o segundo argumento deve ser interpretado como sendo do tipo text.

Concatenação de tipos não especificados:

```
=> SELECT 'abc' || 'def' AS "não especificado";

 não especificado
------------------
 abcdef
(1 linha)
```

Neste caso não existe nenhuma pista inicial do tipo a ser usado, porque não foi especificado nenhum tipo na consulta. Portanto, o analisador procura todos os operadores candidatos, e descobre que existem candidatos aceitando tanto cadeia de caracteres quanto cadeia de bits como entrada. Como a categoria cadeia de caracteres é a preferida quando está disponível, esta categoria é selecionada e, depois, é usado o tipo preferido para cadeia de caracteres, text, como o tipo específico para solucionar os literais de tipo desconhecido.

Exemplo - Resolução do tipo em operador de valor absoluto e negação

O catálogo de operadores do PostgreSQL possui várias entradas para o **operador de prefixo @**, todas **implementando operações de valor absoluto** para vários tipos de dado numéricos. Uma destas entradas é para o tipo float8, que é o tipo preferido da categoria numérica. Portanto, o PostgreSQL usa esta entrada quando na presença de uma entrada não numérica:

```
=> SELECT @ '-4.5' AS "abs";

 abs
-----
 4.5
(1 linha)
```

Aqui o sistema realiza uma conversão implícita de text para float8 antes de aplicar o operador escolhido. Pode ser verificado que foi utilizado float8, e não algum outro tipo, escrevendo-se:

```
=> SELECT @ '-4.5e500' AS "abs";

ERRO:  "-4.5e500" está fora da faixa para o tipo double precision
```

Por outro lado, o **operador de prefixo ~ (negação bit-a-bit)** é definido apenas para tipos de dado inteiros, e não para float8. Portanto, se tentarmos algo semelhante usando ~, resulta em:

```
=> SELECT ~ '20' AS "negação";

ERRO:  operador não é único: ~ "unknown"
DICA:  Não foi possível escolher um operador candidato melhor.
       Pode ser necessário adicionar uma conversão de tipo explícita.
```

Isto acontece porque o sistema não pode decidir qual dos vários **operadores ~** (negativo) possíveis deve ser o preferido. Pode ser dada uma ajuda usando uma conversão explícita:

```
=> SELECT ~ CAST('20' AS int8) AS "negação";

 negação
---------
     -21
(1 linha)
```

Mais detalhes em: http://pgdocptbr.sourceforge.net/pg80/typeconv-oper.html

Precedência dos operadores (decrescente)

Operador/Elemento	Associatividade	Descrição
.	esquerda	separador de nome de tabela/coluna
::	esquerda	conversão de tipo estilo PostgreSQL
[ ]	esquerda	seleção de elemento de matriz
-	direita	menos unário
^	esquerda	exponenciação
* / %	esquerda	multiplicação, divisão, módulo
+ -	esquerda	adição, subtração
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		teste de nulo
NOTNULL		teste de não nulo
(qualquer outro)	esquerda	os demais operadores nativos e os definidos pelo usuário
IN		membro de um conjunto
BETWEEN		contido em um intervalo
OVERLAPS		sobreposição de intervalo de tempo

Operador/Elemento	Associatividade	Descrição
LIKE ILIKE SIMILAR		correspondência de padrão em cadeia de caracteres
< >		menor que, maior que
=	direita	igualdade, atribuição
NOT	direita	negação lógica
AND	esquerda	conjunção lógica
OR	esquerda	disjunção lógica

Detalhes em: http://pgdocptbr.sourceforge.net/pg80/sql-syntax.html#SQL-PRECEDENCE

6.2) Entendendo os Operadores de Texto (Strings)

Funções e operadores para cadeias de caracteres (Strings)

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

O SQL define algumas funções para cadeias de caracteres com uma sintaxe especial, onde certas palavras chave, em vez de vírgulas, são utilizadas para separar os argumentos. Os detalhes estão na Tabela 9-6. Estas funções também são implementadas utilizando a sintaxe regular de chamada de função (Consulte a Tabela 9-7).

Tabela 9-6. Funções e operadores SQL para cadeias de caracteres: vide site oficial em:

http://pgdocptbr.sourceforge.net/pg80/functions-string.html

http://www.postgresql.org/docs/8.3/interactive/functions-string.html

Exemplo 9-1. Conversão de letras minúsculas e maiúsculas acentuadas

Abaixo estão mostradas duas funções para conversão de letras. A função maiusculas converte letras minúsculas, com ou sem acentos, em maiúsculas, enquanto a função minusculas faz o contrário, ou seja, converte letras maiúsculas, com ou sem acentos em minúsculas [1] .

```
=> \!chcp 1252
Active code page: 1252
=> CREATE FUNCTION maiusculas(text) RETURNS text AS '
'>   SELECT translate( upper($1),
'>          text ''áéíóúàèìòùãõâêîôôäëïöüç'',
'>          text ''ÁÉÍÓÚÀÈÌÒÙÃÕÂÊÎÔÛÄËÏÖÜÇ'')
'> ' LANGUAGE SQL STRICT;

=> SELECT maiusculas('à ação seqüência');
```

```
    maiusculas
------------------
 À AÇÃO SEQÜÊNCIA

=> CREATE FUNCTION minusculas(text) RETURNS text AS '
'>   SELECT translate( lower($1),
'>          text ''ÁÉÍÓÚÀÈÌÒÙÃÕÂÊÎÔÛÄËÏÖÜÇ'',
'>          text ''áéíóúàèìòùãõâêîôûäëïöüç'')
'> ' LANGUAGE SQL STRICT;

=> SELECT minusculas('À AÇÃO SEQÜÊNCIA');
    minusculas
------------------
 à ação seqüência
```

6.3) Entendendo as Expressões regulares

Correspondência com padrão (Expressões regulares)

O PostgreSQL disponibiliza três abordagens distintas para correspondência com padrão:

o operador LIKE tradicional do SQL;

o operador mais recente SIMILAR TO (adicionado ao SQL:1999);

e as expressões regulares no estilo POSIX.

Além disso, também está disponível a função de correspondência com padrão substring, que utiliza expressões regulares tanto no estilo SIMILAR TO quanto no estilo POSIX.

> **Dica:** Havendo necessidade de correspondência com padrão acima destas, deve ser considerado o desenvolvimento de uma função definida pelo usuário em Perl ou Tcl.

LIKE

```
cadeia_de_caracteres LIKE padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT LIKE padrão [ESCAPE caractere_de_escape]
```

Cada padrão define um conjunto de cadeias de caracteres. A expressão LIKE retorna verdade se a cadeia_de_caracteres estiver contida no conjunto de cadeias de caracteres representado pelo padrão; como esperado, a expressão NOT LIKE retorna falso quando LIKE retorna verdade, e vice-versa, e a expressão equivalente é NOT (cadeia_de_caracteres LIKE padrão).

Quando o padrão não contém os caracteres percentagem ou sublinhado, o padrão representa apenas a própria cadeia de caracteres; neste caso LIKE atua como o operador igual. No padrão o caractere sublinhado (_) representa (corresponde a) qualquer um único caractere; o caractere percentagem (%) corresponde a qualquer cadeia com zero ou mais caracteres.

Alguns exemplos:

```
'abc' LIKE 'abc'    verdade
'abc' LIKE 'a%'     verdade
'abc' LIKE '_b_'    verdade
'abc' LIKE 'c'      falso
```

Pode ser utilizada a palavra chave ILIKE no lugar de LIKE para fazer a correspondência não diferenciar letras maiúsculas de minúsculas, conforme o idioma ativo. [1] Isto não faz parte do padrão SQL, sendo uma extensão do PostgreSQL.

O operador ~~ equivale ao LIKE, enquanto ~~* corresponde ao ILIKE.

Também existem os operadores !~~ e !~~*, representando o NOT LIKE e o NOT ILIKE respectivamente. **Todos estes operadores são específicos do PostgreSQL**.

Expressões regulares do SIMILAR TO

```
cadeia_de_caracteres SIMILAR TO padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT SIMILAR TO padrão [ESCAPE caractere_de_escape]
```

O operador SIMILAR TO retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é **muito semelhante ao LIKE**, **exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL**. As expressões regulares do padrão SQL são um cruzamento curioso entre a notação do LIKE e a notação habitual das expressões regulares.

Da mesma forma que o LIKE, o operador SIMILAR TO somente é bem-sucedido quando o padrão corresponde a toda cadeia de caracteres; é diferente do praticado habitualmente nas expressões regulares, onde o padrão pode corresponder a qualquer parte da cadeia de caracteres.

Também como o LIKE, o operador SIMILAR TO utiliza _ e % como caracteres curinga, representando qualquer um único caractere e qualquer cadeia de caracteres, respectivamente (são comparáveis ao . e ao .* das expressões regulares POSIX).

Além destas funcionalidades tomadas emprestada do LIKE, o **SIMILAR TO suporta os seguintes metacaracteres para correspondência com padrão pegos emprestado das expressões regulares POSIX:**

- | representa alternância (uma das duas alternativas).
- * representa a repetição do item anterior zero ou mais vezes.
- + representa a repetição do item anterior uma ou mais vezes.
- Os parênteses () podem ser utilizados para agrupar itens em um único item lógico.
- A expressão de colchetes [...] especifica uma classe de caracteres, do mesmo modo que na expressão regular POSIX.

Deve ser observado que as repetições limitadas (? e {...}) não estão disponíveis, embora existam no POSIX. Além disso, **o ponto (.) não é um metacaractere.**

Da mesma forma que no LIKE, a contrabarra desativa o significado especial de qualquer um dos metacaracteres; ou pode ser especificado um caractere de escape diferente por meio da cláusula ESCAPE.

Alguns exemplos:

```
'abc' SIMILAR TO 'abc'      verdade
'abc' SIMILAR TO 'a'        falso
'abc' SIMILAR TO '%(b|d)%'  verdade
'abc' SIMILAR TO '(b|c)%'   falso
```

A função substring com três parâmetros, substring(cadeia_de_caracteres FROM padrão FOR caractere_de_escape), permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular SQL:1999. Assim como em SIMILAR TO, o padrão especificado deve corresponder a toda a cadeia de caracteres, senão a função falha e retorna nulo. Para indicar a parte do padrão que deve ser retornada em caso de sucesso, o padrão deve conter duas ocorrências do caractere de escape seguidas por aspas ("). É retornado o texto correspondente à parte do padrão entre estas marcas.

Alguns exemplos:

```
substring('foobar' FROM '%#"o_b#"%' FOR '#')    oob
substring('foobar' FROM '#"o_b#"%' FOR '#')     NULL
```

Expressões regulares POSIX

A tabela abaixo mostra os operadores disponíveis para correspondência com padrão utilizando as expressões regulares POSIX.

Operadores de correspondência para expressões regulares

Operador	Descrição	Exemplo
~	Corresponde à expressão regular, diferenciando maiúsculas e minúsculas	'thomas' ~ '.*thomas.*'
~*	Corresponde à expressão regular, não diferenciando maiúsculas e minúsculas	'thomas' ~* '.*Thomas.*'
!~	Não corresponde à expressão regular, diferenciando maiúsculas e minúsculas	'thomas' !~ '.*Thomas.*'
!~*	Não corresponde à expressão regular, não diferenciando maiúsculas e minúsculas	'thomas' !~* '.*vadim.*'

As expressões regulares POSIX fornecem uma forma mais poderosa para correspondência com padrão que os operadores LIKE e SIMILAR TO. Muitas ferramentas do Unix, como egrep, sed e awk, utilizam uma linguagem para correspondência com padrão semelhante à descrita aqui.

Uma expressão regular é uma seqüência de caracteres contendo uma definição abreviada de um

conjunto de cadeias de caracteres (um *conjunto regular*). Uma cadeia de caracteres é dita correspondendo a uma expressão regular se for membro do conjunto regular descrito pela expressão regular. Assim como no LIKE, os caracteres do padrão correspondem exatamente aos caracteres da cadeia de caracteres, a não ser quando forem caracteres especiais da linguagem da expressão regular — porém, as expressões regulares utilizam caracteres especiais diferentes dos utilizados pelo LIKE. Diferentemente dos padrões do LIKE, uma expressão regular pode corresponder a qualquer parte da cadeia de caracteres, a não ser que a expressão regular seja explicitamente ancorada ao início ou ao final da cadeia de caracteres.

Alguns exemplos:

```
'abc' ~ 'abc'    verdade
'abc' ~ '^a'     verdade
'abc' ~ '(b|d)'  verdade
'abc' ~ '^(b|c)' falso
```

A função substring com dois parâmetros, substring(cadeia_de_caracteres FROM padrão), permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular POSIX. A função retorna nulo quando não há correspondência, senão retorna a parte do texto que corresponde ao padrão. Entretanto, quando o padrão contém parênteses, é retornada a parte do texto correspondendo à primeira subexpressão entre parênteses (aquela cujo abre parênteses vem primeiro). Podem ser colocados parênteses envolvendo toda a expressão, se for desejado utilizar parênteses em seu interior sem disparar esta exceção. Se for necessária a presença de parênteses no padrão antes da subexpressão a ser extraída, veja os parênteses não-capturantes descritos abaixo.

Alguns exemplos:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

As expressões regulares do PostgreSQL são implementadas utilizando um pacote escrito por Henry Spencer. Grande parte da descrição das expressões regulares abaixo foi copiada textualmente desta parte de seu manual.

Abaixo está mostrado o script usado para criar e carregar a tabela:

```
\!chcp 1252
CREATE TABLE textos(texto VARCHAR(40));
INSERT INTO textos VALUES ('www.apache.org');
INSERT INTO textos VALUES ('pgdocptbr.sourceforge.net');
INSERT INTO textos VALUES ('WWW.PHP.NET');
INSERT INTO textos VALUES ('www-130.ibm.com');
INSERT INTO textos VALUES ('Julia Margaret Cameron');
INSERT INTO textos VALUES ('Sor Juana Inés de la Cruz');
INSERT INTO textos VALUES ('Inês Pedrosa');
INSERT INTO textos VALUES ('Amy Semple McPherson');
INSERT INTO textos VALUES ('Mary McCarthy');
INSERT INTO textos VALUES ('Isabella Andreine');
INSERT INTO textos VALUES ('Jeanne Marie Bouvier de la Motte Guyon');
INSERT INTO textos VALUES ('Maria Tinteretto');
INSERT INTO textos VALUES ('');
INSERT INTO textos VALUES ('  '||chr(9)||chr(10)||chr(11)||chr(12)||chr(13));
INSERT INTO textos VALUES ('192.168.0.15');
INSERT INTO textos VALUES ('pgsql-bugs-owner@postgresql.org');
INSERT INTO textos VALUES('00:08:54:15:E5:FB');
```

A seguir estão mostradas as consultas efetuadas juntamente com seus resultados:

I. Selecionar textos contendo um ou mais caracteres de "a" até "z", seguidos por um ponto, seguido por um ou mais caracteres de "a" até "z", seguidos por um ponto, seguido por um ou mais caracteres de "a" até "z".

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '([a-
z]+).([a-z]+).([a-z]+)';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^([a-z]+)\\.
([a-z]+)\\.([a-z]+)$';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^([a-z]+)[.]
([a-z]+)[.]([a-z]+)$';

           texto
--------------------------
 www.apache.org
 pgdocptbr.sourceforge.net
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto,
'^([a-z]+)\.([a-z]+)\.([a-z]+)$');
-- ou
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto,
'^([a-z]+)[.]([a-z]+)[.]([a-z]+)$');

TEXTO
----------------------------------------
www.apache.org
pgdocptbr.sourceforge.net
WWW.PHP.NET
```

6.4) Entendendo os Operadores matemáticos

São fornecidos operadores matemáticos para muitos tipos de dados do PostgreSQL. Para os tipos sem as convenções matemáticas habituais para todas as permutações possíveis (por exemplo, os tipos de data e hora), o comportamento real é descrito nas próximas seções.

Operadores matemáticos

Operador	Descrição	Exemplo	Resultado
+	adição	2 + 3	5
-	subtração	2 - 3	-1

Operador	Descrição	Exemplo	Resultado
*	multiplicação	2 * 3	6
/	divisão (divisão inteira trunca o resultado)	4 / 2	2
%	módulo (resto)	5 % 4	1
^	exponenciação	2.0 ^ 3.0	8
|/	raiz quadrada	|/ 25.0	5
||/	raiz cúbica	||/ 27.0	3
!	fatorial	5 !	120
!!	fatorial (operador de prefixo)	!! 5	120
@	valor absoluto	@ -5.0	5
&	AND bit a bit	91 & 15	11
|	OR bit a bit	32 | 3	35
#	XOR bit a bit	17 # 5	20
~	NOT bit a bit	~1	-2
<<	deslocamento à esquerda bit a bit	1 << 4	16
>>	deslocamento à direita bit a bit	8 >> 2	2

Os operadores bit a bit trabalham somente em tipos de dado inteiros, enquanto os demais estão disponíveis para todos os tipos de dado numéricos.

A tabela abaixo mostra as funções matemáticas disponíveis. Nesta tabela "dp" significa double precision. Muitas destas funções são fornecidas em várias formas, com diferentes tipos de dado dos argumentos. Exceto onde estiver indicado, todas as formas das funções retornam o mesmo tipo de dado de seu argumento. As funções que trabalham com dados do tipo double precision são, em sua maioria, implementadas usando a biblioteca C do sistema hospedeiro; a precisão e o comportamento em casos limites podem, portanto, variar dependendo do sistema hospedeiro.
Detalhes em: http://pgdocptbr.sourceforge.net/pg80/functions-math.html
http://www.postgresql.org/docs/8.3/interactive/functions-math.html

6.5) Entendendo a importância da Conversão de tipos

Conversão de tipo

Os comandos SQL podem, intencionalmente ou não, usar tipos de dado diferentes na mesma expressão. O PostgreSQL possui muitas funcionalidades para processar expressões com mistura de tipos.
Em muitos casos não há necessidade do usuário compreender os detalhes do mecanismo de conversão de tipo. Entretanto, as conversões implícitas feitas pelo PostgreSQL podem afetar o resultado do comando. Quando for necessário, os resultados podem ser personalizados utilizando

uma conversão de tipo *explícita*.

Este capítulo apresenta os mecanismos e as convenções de conversão de tipo de dado do PostgreSQL.

Visão geral

A linguagem SQL é uma linguagem fortemente tipada, ou seja, todo item de dado possui um tipo de dado associado que determina seu comportamento e a utilização permitida. O PostgreSQL possui um sistema de tipo de dado extensível, muito mais geral e flexível do que o de outras implementações do SQL. Por isso, a maior parte do comportamento de conversão de tipo de dado do PostgreSQL é governado por regras gerais, em vez de heurísticas [1] *ad hoc* [2] , permitindo, assim, expressões com tipos diferentes terem significado mesmo com tipos definidos pelo usuário.

O rastreador/analisador (*scanner/parser*) do PostgreSQL divide os elementos léxicos em apenas cinco categorias fundamentais: inteiros, números não inteiros, cadeias de caracteres, identificadores e palavras chave. As constantes dos tipos não numéricos são, em sua maioria, classificadas inicialmente como cadeias de caracteres. A definição da linguagem SQL permite especificar o nome do tipo juntamente com a cadeia de caracteres, e este mecanismo pode ser utilizado no PostgreSQL para colocar o analisador no caminho correto. Por exemplo, a consulta

```
=> SELECT text 'Origem' AS "local", point '(0,0)' AS "valor";

 local  | valor
--------+-------
 Origem | (0,0)
(1 linha)
```

possui duas constantes literais, dos tipos text e point. Se não for especificado um tipo para o literal cadeia de caracteres, então será atribuído inicialmente o tipo guardador de lugar unknown (desconhecido), a ser determinado posteriormente nos estágios descritos abaixo.

Existem quatro construções SQL fundamentais que requerem regras de conversão de tipo distintas no analisador do PostgreSQL:

Chamadas de função

> Grande parte do sistema de tipo do PostgreSQL é construído em torno de um amplo conjunto de funções. As funções podem possuir um ou mais argumentos. Como o PostgreSQL permite a sobrecarga de funções, o nome da função, por si só, não identifica unicamente a função a ser chamada; o analisador deve selecionar a função correta baseando-se nos tipos de dado dos argumentos fornecidos.

Operadores

> O PostgreSQL permite expressões com operadores unários (um só argumento) de prefixo e de sufixo, assim como operadores binários (dois argumentos). Assim como as funções, os operadores podem ser sobrecarregados e, portanto, existe o mesmo problema para selecionar o operador correto.

Armazenamento do valor

> Os comandos SQL INSERT e UPDATE colocam os resultados das expressões em tabelas. As expressões nestes comandos devem corresponder aos tipos de dado das colunas de destino, ou talvez serem convertidas para estes tipos de dado.

Construções UNION, CASE e ARRAY

> Como os resultados de todas as cláusulas SELECT de uma declaração envolvendo união devem aparecer em um único conjunto de colunas, deve ser feita a correspondência entre os tipos de dado dos resultados de todas as cláusulas SELECT e a conversão em um conjunto uniforme. Do mesmo modo, os resultados das expressões da construção CASE devem ser todos convertidos em um tipo de dado comum, para que a expressão CASE tenha, como um todo, um tipo de dado de saída conhecido. O mesmo se aplica às construções ARRAY.

Os catálogos do sistema armazenam informações sobre que conversões entre tipos de dado, chamadas de *casts*, são válidas, e como realizar estas conversões. Novas conversões podem ser adicionadas pelo usuário através do comando CREATE CAST (Geralmente isto é feito junto com a definição de novos tipos de dado. O conjunto de conversões entre os tipos nativos foi cuidadosamente elaborado, sendo melhor não alterá-lo).

É fornecida no analisador uma heurística adicional para permitir estimar melhor o comportamento apropriado para os tipos do padrão SQL. Existem diversas *categorias de tipo* básicas definidas: boolean, numeric, string, bitstring, datetime, timespan, geometric, network e a definida pelo usuário. Cada categoria, com exceção da definida pelo usuário, possui um ou mais *tipo preferido*, selecionado preferencialmente quando há ambigüidade. Na categoria definida pelo usuário, cada tipo é o seu próprio tipo preferido. As expressões ambíguas (àquelas com várias soluções de análise candidatas) geralmente podem, portanto, serem resolvidas quando existem vários tipos nativos possíveis, mas geram erro quando existem várias escolhas para tipos definidos pelo usuário.

Todas as regras de conversão de tipo foram projetadas com vários princípios em mente:

- As conversões implícitas nunca devem produzir resultados surpreendentes ou imprevisíveis.
- Tipos definidos pelo usuário, para os quais o analisador não possui nenhum conhecimento *a priori*, devem estar "acima" na hierarquia de tipo. Nas expressões com tipos mistos, os tipos nativos devem sempre ser convertidos no tipo definido pelo usuário (obviamente, apenas se a conversão for necessária).
- Tipos definidos pelo usuário não se relacionam. Atualmente o PostgreSQL não dispõe de informações sobre o relacionamento entre tipos, além das heurísticas codificadas para os tipos nativos e relacionamentos implícitos baseado nas funções e conversões disponíveis.
- Não deve haver nenhum trabalho extra do analisador ou do executor se o comando não

necessitar de conversão de tipo implícita, ou seja, se o comando estiver bem formulado e os tipos já se correspondem, então o comando deve prosseguir sem despender tempo adicional no analisador, e sem introduzir chamadas de conversão implícita desnecessárias no comando.

Além disso, se o comando geralmente requer uma conversão implícita para a função, e se o usuário definir uma nova função com tipos corretos para os argumentos, então o analisador deve usar esta nova função, não fazendo mais a conversão implícita utilizando a função antiga.

Notas

[1] *heurística* — Rubrica: informática. — método de investigação baseado na aproximação progressiva de um dado problema. Dicionário Eletrônico Houaiss da língua portuguesa 1.0 (N. do T.)

[2] *ad hoc* — destinado a essa finalidade; feito exclusivamente para explicar o fenômeno que descreve e que não serve para outros casos, não dando margem a qualquer generalização (diz-se de regra, argumento, definição etc.) — Dicionário Eletrônico Houaiss da língua portuguesa 1.0 (N. do T.)

Detalhes em: http://pgdocptbr.sourceforge.net/pg80/typeconv.html

Funções

A função específica a ser utilizada em uma chamada de função é determinada de acordo com os seguintes passos.

Resolução do tipo em função

1. Selecionar no catálogo do sistema pg_proc as funções a serem consideradas. Se for utilizado um nome de função não qualificado, as funções consideradas serão aquelas com nome e número de argumentos corretos, visíveis no caminho de procura corrente (consulte a Seção 5.8.3). Se for fornecido um nome de função qualificado, somente serão consideradas as funções no esquema especificado.
    a. Se forem encontradas no caminho de procura várias funções com argumentos do mesmo tipo, somente será considerada àquela que aparece primeiro no caminho. Mas as funções com argumentos de tipos diferentes serão consideradas em pé de igualdade, não importando a posição no caminho de procura.
2. Verificar se alguma função aceita exatamente os mesmos tipos de dado dos argumentos de entrada. Caso exista (só pode haver uma correspondência exata no conjunto de funções consideradas), esta é usada. Os casos envolvendo o tipo unknown nunca encontram correspondência nesta etapa.
3. Se não for encontrada nenhuma correspondência exata, verificar se a chamada de função parece ser uma solicitação trivial de conversão de tipo. Isto acontece quando a chamada de

função possui apenas um argumento, e o nome da função é o mesmo nome (interno) de algum tipo de dado. Além disso, o argumento da função deve ser um literal de tipo desconhecido, ou um tipo binariamente compatível com o tipo de dado do nome da função. Quando estas condições são satisfeitas, o argumento da função é convertido no tipo de dado do nome da função sem uma chamada real de função.

4. Procurar pela melhor correspondência.
    a. Desprezar as funções candidatas para as quais os tipos da entrada não correspondem, e nem podem ser convertidos (utilizando uma conversão implícita) para corresponder. Para esta finalidade é assumido que os literais do tipo unknown podem ser convertidos em qualquer tipo. Se permanecer apenas uma função candidata, então esta é usada; senão continuar na próxima etapa.
    b. Examinar todas as funções candidatas, e manter aquelas com mais correspondências exatas com os tipos da entrada (Para esta finalidade os domínios são considerados idênticos aos seus tipos base). Manter todas as funções candidatas se nenhuma possuir alguma correspondência exata. Se permanecer apenas uma função candidata, então esta é usada; senão continuar na próxima etapa.
    c. Examinar todas as funções candidatas, e manter aquelas que aceitam os tipos preferidos (da categoria de tipo do tipo de dado de entrada) em mais posições onde a conversão de tipo será necessária. Manter todas as candidatas se nenhuma aceitar o tipo preferido. Se permanecer apenas uma função candidata, esta será usada; senão continuar na próxima etapa.
    d. Se algum dos argumentos de entrada for do tipo "unknown", verificar as categorias de tipo aceitas nesta posição do argumento pelas funções candidatas remanescentes. Em cada posição, selecionar a categoria string se qualquer uma das candidatas aceitar esta categoria (este favorecimento em relação à cadeia de caracteres é apropriado, porque um literal de tipo desconhecido se parece com uma cadeia de caracteres). Senão, se todas as candidatas remanescentes aceitam a mesma categoria de tipo, selecionar esta categoria; senão falhar, porque a escolha correta não pode ser deduzida sem informações adicionais. Rejeitar agora as funções candidatas que não aceitam a categoria de tipo selecionada; além disso, se alguma função candidata aceitar o tipo preferido em uma dada posição do argumento, rejeitar as candidatas que aceitam tipos não preferidos para este argumento.
    e. Se permanecer apenas uma função candidata, este será usada; Se não permanecer nenhuma função candidata, ou se permanecer mais de uma candidata, então falhar.

Deve ser observado que as regras da "melhor correspondência" são idênticas para a resolução do tipo em operador e função. Seguem alguns exemplos.

Exemplo - Resolução do tipo do argumento em função de arredondamento

Existe apenas uma função round com dois argumentos (O primeiro é numeric e o segundo é integer). Portanto, a consulta abaixo converte automaticamente o primeiro argumento do tipo integer para numeric:

```
=> SELECT round(4, 4);

 round
--------
 4.0000
(1 linha)
```

Na verdade esta consulta é convertida pelo analisador em

```
=> SELECT round(CAST (4 AS numeric), 4);
```

Uma vez que inicialmente é atribuído o tipo numeric às constantes numéricas com ponto decimal, a consulta abaixo não necessita de conversão de tipo podendo, portanto, ser ligeiramente mais eficiente:

```
=> SELECT round(4.0, 4);
```

Exemplo - Resolução do tipo em função de subcadeia de caracteres

Existem diversas funções substr, uma das quais aceita os tipos text e integer. Se esta função for chamada com uma constante cadeia de caracteres de tipo não especificado, o sistema escolherá a função candidata que aceita o argumento da categoria preferida para string (que é o tipo text).

```
=> SELECT substr('1234', 3);

 substr
--------
     34
(1 linha)
```

Se a cadeia de caracteres for declarada como sendo do tipo varchar, o que pode ser o caso se vier de uma tabela, então o analisador tenta converter para torná-la do tipo text:

```
=> SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 linha)
```

Esta consulta é transformada pelo analisador para se tornar efetivamente:

```
=> SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Nota:** O analisador descobre no catálogo pg_cast que os tipos text e varchar são binariamente compatíveis, significando que um pode ser passado para uma função que aceita o outro sem realizar qualquer conversão física. Portanto, neste caso, não é realmente inserida nenhuma chamada de conversão de tipo explícita.

E, se a função for chamada com um argumento do tipo integer, o analisador tentará convertê-lo em text:

```
=> SELECT substr(1234, 3);

 substr
--------
     34
(1 linha)
```

Na verdade é executado como:

```
=> SELECT substr(CAST (1234 AS text), 3);
```

Esta transformação automática pode ser feita, porque existe uma conversão implícita de integer para text que pode ser chamada.

Armazenamento de valor

Os valores a serem inseridos na tabela são convertidos no tipo de dado da coluna de destino de acordo com as seguintes etapas.

Conversão de tipo para armazenamento de valor

1. Verificar a correspondência exata com o destino.
2. Senão, tentar converter a expressão no tipo de dado de destino. Isto será bem-sucedido se houver uma conversão registrada entre os dois tipos. Se a expressão for um literal de tipo desconhecido, o conteúdo do literal cadeia de caracteres será enviado para a rotina de conversão de entrada do tipo de destino.
3. Verificar se existe uma conversão de tamanho para o tipo de destino. Uma conversão de tamanho é uma conversão do tipo para o próprio tipo. Se for encontrada alguma no catálogo pg_cast aplicá-la à expressão antes de armazenar na coluna de destino. A função que implementa este tipo de conversão sempre aceita um parâmetro adicional do tipo integer, que recebe o comprimento declarado da coluna de destino (na verdade, seu valor atttypmod; a interpretação de atttypmod varia entre tipos de dado diferentes). A função de conversão é responsável por aplicar toda semântica dependente do comprimento, tal como verificação do tamanho ou truncamento.

Exemplo - Conversão de tipo no armazenamento de *character*

Para uma coluna de destino declarada como character(20), a seguinte declaração garante que o valor

armazenado terá o tamanho correto:

```
=> CREATE TABLE vv (v character(20));
=> INSERT INTO vv SELECT 'abc' || 'def';
=> SELECT v, length(v) FROM vv;

          v           | length
----------------------+--------
 abcdef               |     20
(1 linha)
```

O que acontece realmente aqui, é que os dois literais desconhecidos são resolvidos como text por padrão, permitindo que o operador || seja resolvido como concatenação de text. Depois, o resultado text do operador é convertido em bpchar ("caractere completado com brancos", ou "*blank-padded char*", que é o nome interno do tipo de dado character) para corresponder com o tipo da coluna de destino (Uma vez que os tipos text e bpchar são binariamente compatíveis, esta conversão não insere nenhuma chamada real de função). Por fim, a função de tamanho bpchar(bpchar, integer) é encontrada no catálogo do sistema, e aplicada ao resultado do operador e comprimento da coluna armazenada. Esta função específica do tipo realiza a verificação do comprimento requerido, e adiciona espaços para completar.

Construções UNION, CASE e ARRAY

As construções UNION do SQL precisam unir tipos, que podem não ser semelhantes, para que se tornem um único conjunto de resultados. O algoritmo de resolução é aplicado separadamente a cada coluna de saída da consulta união. As construções INTERSECT e EXCEPT resolvem tipos não semelhantes do mesmo modo que UNION. As construções CASE e ARRAY utilizam um algoritmo idêntico para fazer a correspondência das expressões componentes e selecionar o tipo de dado do resultado.

Resolução do tipo em *UNION*, *CASE* e *ARRAY*

1. Se todas as entradas forem do tipo unknown, é resolvido como sendo do tipo text (o tipo preferido da categoria cadeia de caracteres). Senão, ignorar as entradas unknown ao escolher o tipo do resultado.
2. Se as entradas não-desconhecidas não forem todas da mesma categoria de tipo, falhar.
3. Escolher o primeiro tipo de entrada não-desconhecido que for o tipo preferido nesta categoria, ou que permita todas as entradas não-desconhecidas serem convertidas implicitamente no mesmo.
4. Converter todas as entradas no tipo selecionado.

Seguem alguns exemplos.

Exemplo - Resolução do tipo com tipos subespecificados em uma união

```
=> SELECT text 'a' AS "texto" UNION SELECT 'b';

 texto
-------
 a
```

```
 b
(2 linhas)
```

Neste caso, o literal de tipo desconhecido 'b' é resolvido como sendo do tipo text.

Exemplo - Resolução do tipo em uma união simples

```
=> SELECT 1.2 AS "numérico" UNION SELECT 1;

 numérico
----------
        1
      1.2
(2 linhas)
```

O literal 1.2 é do tipo numeric, e o valor inteiro 1 pode ser convertido implicitamente em numeric, portanto este tipo é utilizado.

Exemplo - Resolução do tipo em uma união transposta

```
=> SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 linhas)
```

Neste caso, como o tipo real não pode ser convertido implicitamente em integer, mas integer pode ser implicitamente convertido em real, o tipo do resultado da união é resolvido como real.